Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Hannu Havila

Järjestelmäriippumattomien Flutter- ja React Native -ohjelmistokehysten vertailu

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Ohjelmistotuotanto

Insinöörityö

16.10.2019

Tekijä Otsikko Sivumäärä Aika	Hannu Havila Järjestelmäriippumattomien Flutter- ja React Native -ohjelmistokehysten vertailu 49 sivua + 2 liitettä 16.10.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander

Tämän insinöörityön tarkoituksena on verrata ja tutkia kahta järjestelmäriippumatonta ohjelmistokehystä keskenään sekä tutkia, miten järjestelmäriippumaton ja natiivi kehitys poikkeavat toisistaan. Insinöörityöhön valitut teknologiat ovat Googlen kehittämä Flutter sekä Facebookin kehittämä React Native. Molemmat teknologiat edustavat järjestelmäriippumattomia ohjelmistokehyksiä, mutta eroavat kuitenkin toisistaan monin tavoin.

Työssä etsitään vastausta seuraavaan kysymykseen: ”Onko järjestelmäriippumaton kehitysmenetelmä parempi vaihtoehto kuin natiivi kehitysmenetelmä?” Edeltävän kysymyksen lisäksi insinöörityön aikana luodaan molemmilla valituilla teknologioilla käyttöliittymät ja verrataan molempien kehitysprosessia ja oppimiskäyrää.

React Native on Facebookin kehittämä järjestelmäriippumaton ohjelmistokehys, joka on erittäin suosittu yritysten, kehittäjien ja yhteisön keskuudessa. Google on julkaissut kilpailevan ohjelmistokehyksen nimeltään Flutter, joka kasvattaa suosiotaan ja yhteisöä vuosi vuodelta enemmän.

Työn tarkoituksena on myös toimia lukijalle oppaana, jotta lukija voisi suorittaa työssä kehitetyt käyttöliittymät molemmilla teknologioilla. Työssä käydään läpi kaikkien käyttöliittymien sisältämien näkymien koodia.

Ohjelmistoriippumaton kehitys on tullut erittäin pitkälle siitä, mitä se joskus on ollut. Joissakin tapauksissa sovelluksien kehitys järjestelmäriippumattomalla tavalla on parempi vaihtoehto kuin natiivikehitys. Mutta ottaen huomioon, miten järjestelmäriippumattomat kehykset kehittyvät koko ajan ja miten paljon enemmän ne tarjoavat kuin natiivikehitys, voisi sanoa, että järjestelmäriippumaton kehitys on parempi vaihtoehto kuin natiivikehitys.

Järjestelmien kehitysvertailussa huomattiin Flutterin olevan aluksi vaikeampi ohjelmistokehys kehittää, mutta lopuksi se olikin hyvin yksinkertainen React Nativeen verrattuna.

Avainsanat	Flutter, React Native, Järjestelmäriippumaton kehitys

Author Title Number of Pages Date	Hannu Havila Flutter vs React Native, Comparison Between Cross-Platform Frameworks 49 pages + 2 appendices 16 October 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer

The purpose of this thesis is to compare and investigate two cross-platform frameworks between each other and see how cross-platform development differs from native development. The chosen technologies are Flutter developed by Google and React Native developed by Facebook. Both of the technologies are considered as cross-platform mobile development frameworks, but they also differ quite a lot from each other.

This thesis tries to answer the following question “Are cross-platform development frameworks better for application development than native development methods?”. In addition, mobile applications with both technologies were created during the project and the development process and learning curve during development were compared.

React Native is a cross-platform framework developed by Facebook. It is quite popular with companies, developers and community. Flutter was launched by Google to compete with React Native. It is constantly growing its popularity amongst developers and community.

This thesis servers as a guide for the reader develop similar applications with the both technologies. The study goes through all screens and how to develop them.

Cross-platform development has come a long way from what it used to be. In some situations, the cross-platform development is a better way to go than native development, considering how cross-platform frameworks get better and better and how much more they offer than native development.

During the development of the applications, it was noticed that Flutter seemed to be harder to develop than React Native at first. But in the end Flutter turned out to be easier and simpler than React Native.

Keywords	Flutter, React Native, Cross-platform development

Sisällys

1 Johdanto

Insinöörityön tarkoituksena on verrata ja tutkia kokeellisesti, miten kahden erilaisen järjestelmäriippumattoman ohjelmistokehyksen toiminnot ja kehitysmenetelmät eroavat toisistaan. Vertailussa tutustutaan aluksi molempien teknologioiden perustietoihin, jotta kokeellinen tutkimus voitaisiin suorittaa. Lisäksi työn tarkoituksena on tutkia, miten järjestelmäriippumaton (ohjelma, joka voi toimia useammalla kuin yhdellä käyttöjärjestelmällä) sekä natiivi (ohjelma, joka on kehitetty vain tietylle käyttöjärjestelmälle) mobiilisovelluskehitys poikkeavat toisistaan.

Työn tavoitteena on saavuttaa perusosaaminen molemmista teknologioista, kyky luoda molemmilla ohjelmistokehyksillä käyttöliittymä sekä vastata seuraavaan kysymykseen:

- Onko järjestelmäriippumaton kehitysmenetelmä parempi vaihtoehto kuin natiivi kehitysmenetelmä?

Insinöörityö aloitetaan tutustumalla järjestelmäriippumattomiin ja natiiveihin kehitysmenetelmiin, jonka jälkeen tutkin työhön valittuja teknologioita. Teknologioita ovat Googlen kehittämä ohjelmistokehityspaketti eli ohjelmistokehys nimeltään Flutter sekä Facebookin kehittämä ohjelmistokehys nimeltään React Native. Kyseiset teknologiat valittiin työhön, koska molemmat edustavat avoimen lähdekoodin ohjelmistokehyksiä, mutta eroavat toisistaan koodikieleltään, rakenteiltaan sekä toiminnoiltaan. Insinöörityön alussa tutustutaan teknologioiden teoriaan. Tämän jälkeen suoritan kokeellisen testin, jossa luon molemmilla ohjelmistokehyksillä mobiilisovelluksen. Sovelluksen tulee sisältämään useita näkymiä (Login, Register, Home, Search, Profile, jne.) sekä molempiin testisovelluksiin luodaan navigoimiseen tarkoitetut elementit.

2 Natiivin ja järjestelmäriippumattoman kehityksen konseptin ymmärtäminen

Ennen varsinaista työtä, jossa rakennetaan tehtävään valituista teknologioista testi- ja vertailukäyttöliittymät, täytyy ymmärtää, miten natiivi- ja järjestelmäriippumattomien so-

velluksien kehitykset eroavat toisistaan. Tässä luvussa kerrotaan, mitä järjestelmäriippumattomat ohjelmistokehykset (cross-platform application frameworks) ovat ja miten ne toimivat.

2.1 Mikä on järjestelmäriippumaton ohjelmistokehys?

Järjestelmäriippumattomat ohjelmistokehykset sallivat kehittäjien luoda mobiilisovelluksia, jotka ovat yhteensopivia useamman kuin yhden käyttöjärjestelmän kanssa. Tässä tapauksessa tarkoitetaan iOS- ja Android-järjestelmiä. Järjestelmäriippumattomat ohjelmistokehykset tarjoavat kehittäjille mahdollisuuden kirjoittaa ohjelmistokoodin vain kerran ja sen jälkeen ajaa kyseisen koodin myös eri käyttöjärjestelmillä. Kun ensimmäisiä edellä mainittuja kehyksiä julkaistiin, oli niissä aluksi ongelmia suorituskyvyssä sekä virheellistä sovelluskäyttäytymistä. Nykyisin nämä kehykset ovat kuitenkin kehittyneet ja niistä on tullut yleisiä, koska kummankin alustan täysin natiivien ja alkuperäisten sovelluksien kehittämisen kustannukset nousevat päivä päivältä. (Manchanda 2019.)

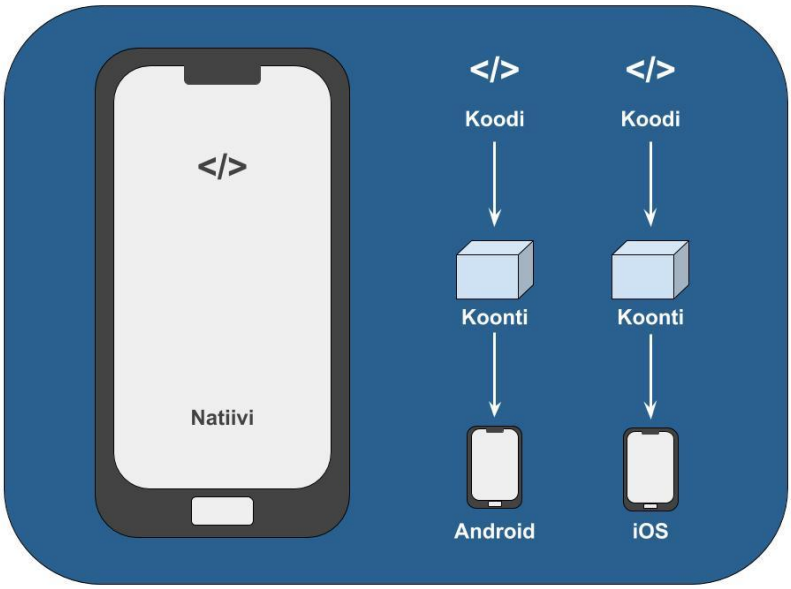
2.2 Ero järjestelmäriippumattoman ja natiivikehityksen välillä

Sovelluskäyttäjien näkökulmasta ei ole juurikaan väliä, onko käytössä oleva sovellus kehitetty natiivi- vai järjestelmäriippumattomien menetelmien kautta. Käyttäjät kokevat enimmäkseen vain sovelluksen käyttöliittymän käytettävyyden ja ulkonäön. Tästä syystä väittely siitä, pitäisikö sovellukset kehittää käyttämällä järjestelmäriippumattomia kehyksiä vai ei, käydään enimmäkseen kehittäjien ja yritysten välillä. (Manchanda 2018.)

Suurimmat erot, joita yritykset yleensä ajattelevat, liittyvät lähinnä tuotekehityskustannuksiin ja tuotteen kehittämiseen vaadittavaan aikaan sekä siihen, kuinka tuote toimii valittujen laitteiden käyttöjärjestelmien ja sen versioiden kanssa (Manchanda 2018).

2.3 Natiivikehitys

Jos kehitettävä ohjelmisto on päätetty kehittää vain tietylle mobiilikäyttöjärjestelmälle, sen määritellään silloin olevan ”puhdas” natiivisovellus. Kehitettävissä natiivimobiilisovelluksissa käytetään ohjelmointikielenä Javaa sekä Kotlinia Android-alustoille ja Objective C:tä tai Swiftiä iOS-alustoille. Koska maailmanlaajuiset mobiilimarkkinat ovat Android- ja iOS-painotteisia (statcounter 2019) haluavat monet yritykset sekä kehittäjät tuottaa omat sovelluksensa vain näille alustoille. Natiivisovelluksia kehittäessä sovellukset pystyvät käyttämään käyttöjärjestelmän sisäänrakennettuja turvallisuusominaisuuksia toisin kuin järjestelmäriippumattomat sovellukset. Kuitenkin natiivisovelluksien turvallisuusmenetelmät voidaan murtaa huonosti suojatulla paikallisesti tallennetulla datalla, huonosti toteutetulla SSL (Secure Sockets Layer, turvallisuus protokolla, joka luo salatun yhteyden käyttäjän ja palvelimen välille) -implementaatiolla ja tietojen vuotamisella. (Manchanda 2018.)



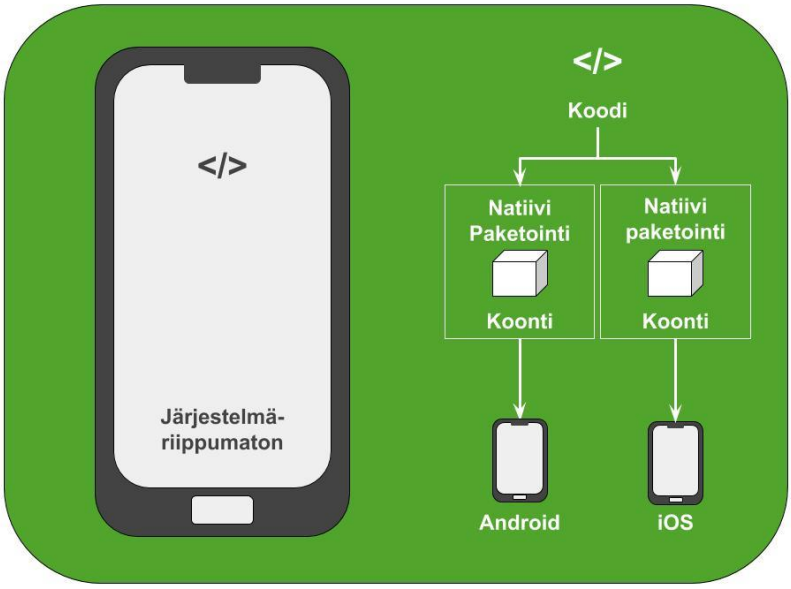
Kuva 1. Esimerkki kehityksen kulusta natiivilla kehitysmenetelmällä.

Kuvassa yksi esitetään, miten natiivilla kehitysmenetelmällä joudutaan ylläpitämään useampaa koodikantaa, jos toteutettava sovellus halutaan julkaista useammalle käyttöjär-

jestelmälle toimivaksi. Tämä kehitysmenetelmä lisää ohjelmiston tuottajan kuluja koodikannan ylläpitämisen ja tuottamisen kannalta, mutta luo usein paljon stabiilimman ohjelmiston.

2.4 Järjestelmäriippumaton kehitys

Jos kehitettävä ohjelmisto on päätetty kehittää useammalle käyttöjärjestelmälle toimivaksi, on usein valintana järjestelmäriippumaton ohjelmistokehys (Manchanda 2018). Useammat kehykset on rakennettu niin, että ne käyttävät HTML5:ttä, CSS:ää ja JavaScriptiä ohjelmien kehitysvaiheessa. Ohjelmaa käännettäessä koodi, joka on tuotettu, ”paketoidaan” natiivikoodin sisälle. Koodin ”paketoiminen” tekee koodista yhteensopivan useamman alustan kanssa. Tavallaan ohjelmistot, jotka on tuotettu järjestelmäriippumattomalla tavalla, operoivat itseään ikään kuin olisivat natiiveja sovelluksia, mutta samankaltaisuus rajoittuu sovelluksen kehitysvaiheessa käytettyyn ohjelmistokehykseen. Esimerkiksi insinöörityössä käytettävä React Native ”keskustelee” natiivisäikeiden kanssa, kun taas esimerkiksi Flutter kääntää koodin täysin natiiviksi. Usein ohjelmistokehykset kuitenkin rajoittavat sovellusta ja sisältävät ainoastaan perushallintaliikkeet, navigointielementit sekä muut olennaiset toiminnot. Järjestelmäriippumattomat ohjelmat pystyvät käyttämään laitteen koneistoa kuten GPS-palveluita, kameraa, osoitekirjaa ja monia muita. Kuitenkaan käytettävyys ei välttämättä aina ole täysin sama kuin ohjelmassa, joka on tuotettu natiivina. (Manchanda 2018.)



Kuva 2. Esimerkki kehityksen kulusta järjestelmäriippumattomalla kehitysmenetelmällä.

Kuvassa kaksi esitetään, miten järjestelmäriippumattomalla kehitysmenetelmällä voidaan luoda vain yhdellä koodikannalla tuotettava sovellus. Koodikanta paketoidaan ja emuloidaan koontivaiheessa yhteensopivaksi jokaisen halutun käyttöjärjestelmän kanssa. Tämä kehitysmenetelmä vähentää sovelluksen tuottajan kuluja huomattavasti sekä nopeuttaa usein sovelluksen toimitusta asiakkaille. Mutta järjestelmäriippumattomalla kehitysmenetelmällä voidaan saada aikaiseksi epästabiilimpi ohjelmisto ja törmätä yhteensopivuusongelmiin myöhemmässä vaiheessa.

3 Teknologioiden ymmärtäminen

Ennen kuin voi kirjoittaa riviäkään koodia testikäyttöliittymiä varten on tärkeää ymmärtää joitakin perusasioita kahdesta työssä verrattavasta teknologiasta. Tässä luvussa käydään läpi perusteoriaa sekä tietoja valituista teknologioista ja kerrotaan kaikki tarvittava, jotta testikäyttöliittymien kehittäminen näillä teknologioilla onnistuisi mahdollisimman helposti.

3.1 React Native

React Native on Facebookin kehittämä avoimen lähdekoodin ohjelmistokehys, jolla on mahdollista tehdä nimensä mukaisia ”natiiveja” sovelluksia Android-, iOS-, Web- ja UWP-alustoille. React Native on hyvin samanlainen kuin React, mutta se käyttää sisäänrakennettuja valmiita natiivikomponentteja rakennuspalikoinaan, kun taas Reactissa komponentit koostuvat kokoelmista HTML-, CSS- ja JavaScript-koodia. Jotta voisi ymmärtää React Native -ohjelman perusrakenteen on ensin ymmärrettävä joitakin perus-React-konsepteja. Näihin konsepteihin lukeutuu muun muassa. JSX-syntaksilaajennos, React-komponentit (components), tilat (state), ja syötteet (props). Vaikka ymmärtäisikin jo jotakin React-kehittämisestä, on silti vielä opittava joitakin vain React Nativelle tyypillisiä asioita kuten natiivit komponentit (native components). (reactjs.org a.)

3.1.1 JSX-syntaksilaajennos

Reactissa ja React Nativessa renderöintilogiikka liittyy luonnostaan muihin käyttöliittymä logiikoihin, esimerkiksi miten käyttöliittymän tapahtumia käsitellään, miten tilat (states) muuttuvat ja miten data valmistellaan näytölle. Sen sijaan, että käyttöliittymän rakenne ja sen sisäinen logiikka eriteltäisiin erillisiin tiedostoihin, niin React yhdistää rakenteen ja logiikan JSX-syntaksilaajennoksella. JSX on syntaksilaajennos JavaScript-ohjelmointikieleen. Se voi muistuttaa ulkonäöltään hyvin paljon HTML-koodia tai normaalin string-muuttujan määrittelyä, mutta se ei ole kumpaakaan. React- ja React Native -komponentit kirjoitetaan usein käyttämällä JSX-syntakseja (ks. esimerkkikoodi 1), mutta tämä ei ole pakollista, vaan komponentit voidaan myös kirjoittaa käyttämällä täysin puhdasta JavaScript-koodia. (reactjs.org a.)

```
const myElement = <div><h1>This is JSX</h1></div>

function App() {
     return (
          myElement
     );
}
```

Esimerkkikoodi 1. JSX-syntaksi, joka määrittelee elementin ”myElement” sisältävän div-elementin sekä h1-luokan tekstin. Tämä elementti kutsutaan funktiossa ja se renderöidään näytölle.

3.1.2 React-komponentit (components)

Jotta voisi kehittää helposti React Native -komponentteja on suositeltavaa ensin ymmärtää perusteet React-komponenteista, sillä React Nativen kaikki toiminnot perustuvat Reactiin. React-komponentit ovat konseptiltaan samanlaisia kuin JavaScript-funktiot. Ne hyväksyvät syötteitä (joita kutsutaan nimellä ”props”) ja palauttavat React-elementin, joka määrittää, mitä näytöllä pitäisi tapahtua. Komponentin voi helpoiten luoda käyttämällä yksinkertaista JavaScript-funktiota, joka vastaanottaa syötteen ja palauttaa lopuksi elementin, jossa tulostetaan vastaanotettu syöte (ks. esimerkkikoodi 2).

```
function Home(props) {
     return <div><h1>This is Home</h1><p>Welcome, {props.username}</p></div>;
}
```

Esimerkkikoodi 2. React-komponentin luominen käyttämällä normaalia JavaScript-funktiota. Funktiota kutsuttaessa funktio palauttaa div-elementin, joka sisältää h1-luokan tekstin sekä tekstin, jossa tervehditään käyttäjää.

Edellä esitetty tapa on hyvin nopea keino luoda komponentti, jos sellaiselle on tarvetta. Lisäksi funktionaalinen komponentti antaa suorituskykyä luotavalle ohjelmistolle. Komponentti voidaan myös luoda käyttämällä ES6-luokkaa. ES6-luokalla tarkoitetaan ECMA-skripti ohjelmointikieltä. ECMA-skripti on standardoitu nimitys JavaScriptille.

```
class Home extends React.Component {
  render() {
    return (
      <div>
        <h1>This is Home</h1>
        <p>Welcome, {this.props.username}</p>
      </div>
    );
  }
}
```

Esimerkkikoodi 3. React-komponentin luominen käyttämällä ES6-luokkaa. Koodi palauttaa identtisen elementin kuin esimerkkikoodi 2.

ES6-luokalla luotu komponentti (ks. esimerkkikoodi 3) vaatii kehittäjää laajentamaan komponenttia React.Component-komponentilla ja luomaan renderöintifunktion. Tämä tapa vaatii enemmän koodia, mutta luo samalla monia etuja verrattuna puhtaaseen JavaScript-funktiolla luotuun komponenttiin. Jos komponentit luodaan puhtaalla JavaScrip-

tillä, ei komponentissa voida käyttää Reactin setState-funktiota, jolloin näitä komponentteja kutsutaan nimellä funktionaaliset tilattomat komponentit (functional stateless components). (Jöch 2018.)

3.1.3 React Native -komponentit (React Native components)

React Native -komponenttien tarkoituksena on olla komponentteja, jotka vastaavat mobiiliympäristön alkuperäisiä käyttöjärjestelmäkomponentteja. React Native tarjoaa kehittäjälle useita erilaisia sisäänrakennettuja komponentteja, jotka toimivat useiden eri käyttöjärjestelmien kanssa (ks. esimerkkikoodi 4). Kehitettävään sovellukseen voidaan myös sisällyttää muiden kehittäjien tekemiä kolmannen osapuolen kirjastoja, joilla voidaan kasvattaa käytettävissä olevien komponenttien määrää. (facebook.github.)

```
class Home extends React.Component {
  render() {
    return (
      <View>
        <Text>This is Home</Text>
        <Text>Welcome, {this.props.username}</Text>
      </View>
    );
  }
}
```

Esimerkkikoodi 4. React Native -komponentin luominen. Koodi on samanlainen ja tuottaa saman tuloksen kuin esimerkkikoodi 3, mutta käytetään vain React Nativen tarjoamia komponentteja.

3.1.4 React-tilat (state)

React- ja React Native -ohjelmia rakentaessa on tyypillistä törmätä tilanteeseen, jossa täytyy päivittää esimerkiksi komponentin elementtien sisältöä. Komponentti voidaan luoda kokonaan uudestaan ja ladata edellisen komponentin tilalle, mutta tämä ei ole järkevää ohjelman koodinhallinnan sekä ohjelman suorituskyvyn kannalta. Tällöin voidaan käyttää Reactin state-konseptia, jossa määritellään komponentille tiloja, joita vaihdetaan tarpeen tullen, kuten seuraavassa esimerkkikoodissa tehdään.

```
class Home extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      showText: false,
```

```
    };
  }
  render() {
    return (
      <div>
        {this.state.showText ? <h1>Sample Test</h1> : null}
        <button onClick={() => this.setState({ showText: true })>
         Change State
        </button>
       </div>
    );
  }
}
```

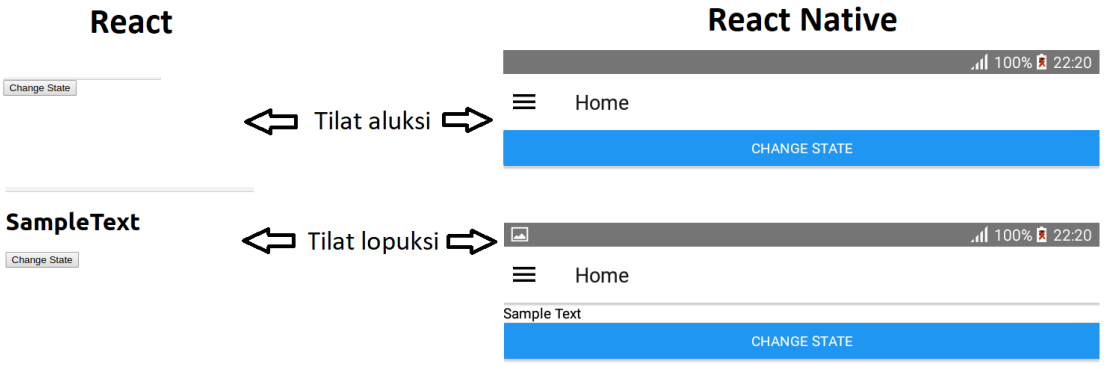
Esimerkkikoodi 5. React-tilan vaihtoesimerkki, jossa määritellään aloitustila, jossa tekstiä ei näytetä. Käyttäjän painaessa nappia ”Change State” -näyttöön ilmestyy teksti ”Sample Test”.

Esimerkkikoodi viisi voidaan käyttää myös React Nativella vaihtamalla muutama HTML-elementti React Native -elementiksi (ks. esimerkkikoodi 6).

```
class Home extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      showText: false,
    };
  }
  render() {
    return (
      <View>
        {this.state.showText ? <Text>Sample Test</Text> : null}
        <Button
          title=”Change State”
          onPress={() => this.setState({ showText: true })
        />
       </View>
    );
  }
}
```

Esimerkkikoodi 6. React Native -tilan muutos.

Kuten huomata saattaa, koodit näyttävät täysin samoilta, mutta erilaisuuksia löytyy elementtien määrittämisessä sekä tietyissä tapahtumakutsuissa. Esimerkiksi React-napin kutsu on ”onClick”, kun taas React Native -napin kutsu on ”onPress”.



Kuva 3. Tilan vaihtamisen lopputulos Reactilla sekä React Nativella.

Koodiesimerkit suorittavat samat toiminnot tilanvaihtamisessa, kuten kuvassa kolme esitetään. Ainoastaan eroina näkyvät käyttöliittymän elementtien tyylit. Näitä muuttamalla esimerkeistä voitaisiin saada identtisen näköiset.

3.1.5 React-syötteet (props)

React-propsit toimivat syötteinä komponenttien ja niiden funktioiden välillä. Props voi olla primitiivi- tai objektidatatyyppi. Primitiivinen data ei ole objekti eikä sillä ole omia metodeja. Kaikki primitiiviset datatyypit ovat muuttumattomia. Objektidatatyyppi säilyttää osoitteita, jotka viittaavat objekteihin. Primitiivisiin datatyyppeihin lukeutuvat:

- totuusarvo (boolean) – true tai false
- tyhjä (null) – ei arvoa
- määrittelemätön (undefined) – esitelty muuttuja, jolle ei ole annettu arvoa
- numero (number) integer, float ja jne…
- merkkijono (string) – taulukko kirjaimia eli sanoja
- symboli (symbol) – uniikki arvo, joka ei vastaa mitään muuta arvoa.

Objektidatatyyppeihin lukeutuvat:

- objekti (object)
- taulukko (Array)

3.2 Flutter

Flutter on Googlen luoma avoimen lähdekoodin ohjelmistokehityspaketti (software development kit eli SDK) ja ohjelmistokehys. Kyseinen paketti on tarkoitettu käytettäväksi sovelluksien kehitykseen Android-, iOS-, Windows-, Mac-, Linux-, Google Fuchsia- ja web-alustoille. Ohjelmistokehyksen ensimmäinen stabiili versio eli 1.0 julkaistiin 4. joulukuuta vuonna 2018. Kehyksestä oli myös julkaistu jo toukokuussa vuonna 2017 Alpha versio 0.0.6. (dart.dev a.)

Kehys on kehitetty käyttämällä C-, C++- ja Dart-ohjelmointikieliä (GitHub a) sekä Skia Graphics Engineä (GitHub b). Flutterilla kehittäessä sovelluksia on kielenä Dart. Flutter-ohjelmistokehys toimii New BSD-lisenssin alaisena (GitHub c).

3.2.1 Dart-ohjelmointikieli

Dart on Googlen kehittämä ohjelmointikieli, joka on luokkapohjainen, objekti-painotteinen dynaaminen kieli, joka myös hoitaa ”roskien” keräämisen automaattisesti ohjelmoinnin aikana (Ladd 2012; Sullivan 2019). Dart-ohjelmoinnissa on monia konsepteja, joita kehittäjien tulisi muistaa noudattaa. (dart.dev b.) Tärkeimmät konseptit ovat:

> Kaikki, mitä kehittäjä sijoittaa muuttujiin ovat objekteja ja jokainen objekti on luokan instanssi. Näihin lukeutuvat myös funktiot, numerot ja tyhjät (null) muuttujat. Kaikki objektit periytyvät Object-luokasta. (dart.dev b)
>
> Vaikka Dart-ohjelmointikieli on vahvasti tyypitetty, ovat silti tyyppien määrittelyt vaihtoehtoisia, sillä Dart pystyy päättelemään muuttujien tyypit tarvittaessa. (dart.dev b)
>
> Dart tukee geneerisiä tyyppejä esimerkiksi listaa kokonaisluvuista tai listaa mistä tahansa objektityypistä. (dart.dev b)
>
> Dart tukee ylimmän tason funktioita, kuten esimerkiksi main(). Tuettuihin funktioihin lukeutuvat myös funktiot, jotka ovat sidottu luokkaan tai objekteihin. Näiden lisäksi kehittäjä voi luoda sisäkkäisiä funktioita. (dart.dev b)
>
> Dart-ohjelmointikielessä ei ole avainsanoja public, protected ja private, vaan jos tunnus alkaa alaviivalla, on silloin muuttuja yksityinen sen kirjastolle. (dart.dev b)

> Tunnukset voivat alkaa kirjaimella tai alaviivalla ja jatkuvat millä tahansa yhdistelmällä kirjaimia sekä numeroita. (dart.dev b)

> Dartilla on sekä lausekkeita (joilla on suoritusaika-arvot) että lauseita (joilla ei ole). Lause usein sisältää useamman kuin yhden lausekkeen, mutta määritelmä ei voi suoraan sisältää lausetta. (dart.dev b)

> Dart-työkalut voivat ilmoittaa kehittäjälle kahdenlaisia ongelmia, jotka ovat varoitukset (warnings) ja virheet (errors). Varoitukset ovat vain indikaatioita siitä, että kehitteillä oleva koodi ei välttämättä toimi, mutta tämä ei estä kehittäjää ajamasta omaa koodiaan. Virheet voivat olla joko koonninaikaisia tai ajonaikaisia virheitä. Koonninaikaiset virheet estävät koodia suorittamasta itseään kokonaan ja ajonaikainen virhe ei estä ohjelman suorittamista, mutta ilmoittaa siitä kehittäjälle koodin ajamisen aikana. (dart.dev b)

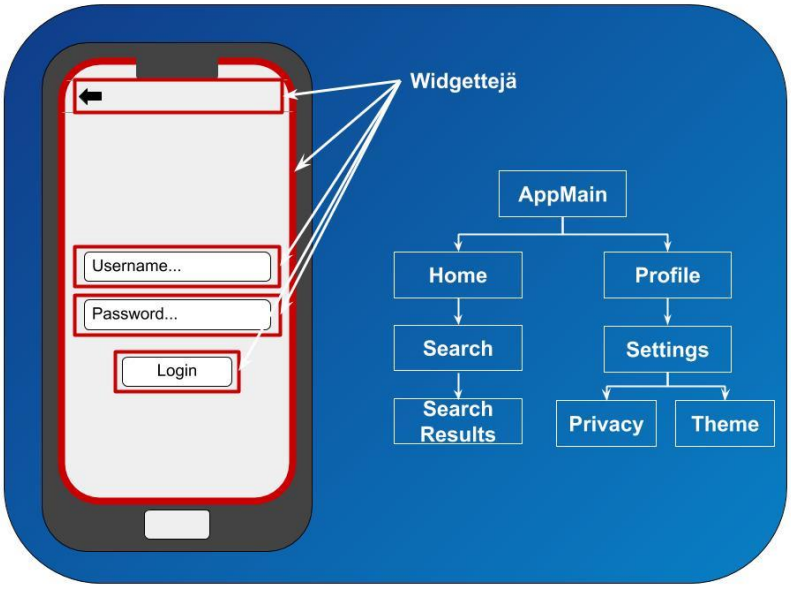
Dart-ohjelmointikieli sisältää myös monien muiden ohjelmistokielien tapaan avainsanoja, joita tulisi välttää käyttämästä tunnuksina. Tarpeen vaatiessa kehittäjä voi käyttää joitakin avainsanoja tunnuksina. Avainsanat lajitellaan Dart-kielessä kolmeen käytettävään luokkaan sekä yhteen varattuun luokkaan. (dart.dev b.) Dart-ohjelmointikieli, jota Flutter ohjelmistokehys käyttää ohjelmien rakentamiseen, on suunniteltu helposti omaksuttavaksi kehittäjille, joilla on kokemusta C#-, Java-, ActionScript- ja JavaScript-ohjelmointikielistä. (Ladd 2012.)

3.2.2 Flutter SDK ja Dart

Flutter-ohjelmistokehityspaketti ei ole sama asia kuin Dart-ohjelmointikieli, vaan Flutter on erillinen ohjelmisto, jota Dart-ohjelmointikieli käyttää. Flutter tarjoaa kehittäjälle ainoastaan valmiit työkalut ja widgetit. Lisäksi Flutter SDK sisältää kääntäjän, joka muuntaa Dart-koodin Androidille ja iOS-järjestelmien natiiviksi koodiksi.

3.2.3 Flutter-ohjelmistojen arkkitehtuuri

Kehitettäessä sovelluksia tai ohjelmistoa Flutterilla on ymmärrettävä, että jokainen elementti, joka ohjelmistoon koodataan, toimii widgettinä. Widgetit ovat Flutter-sovelluksen rakennuspalikat, joista lopullinen ohjelmisto tulee koostumaan. Jokainen widgetti on muuttumaton osa käyttöliittymää. Flutter eroaa toisista järjestelmäriippumattomista ohjelmistokehyksistä käyttämällä vain yhtä konseptia, joka on edellä mainittu widgetti-konsepti, toisin kuin muut ohjelmistokehykset, jotka jakavat osiin käyttöliittymien näkymät, niiden toiminnallisuudet, layoutit ja muut olennaiset osat. (flutter.dev b.)



Kuva 4. Havainnollistaminen miten Flutterilla rakennetun ohjelmiston konsepti toimii. Jokainen elementti on widgetti ja jopa itse sivu ja sovellus lasketaan widgeteiksi. Jokainen ohjelma, mikä tuotetaan Flutterilla voidaan kuvitella olevan puurakenne, joka on rakennettu pelkästään widgeteistä.

Yllä oleva kuva havainnollistaa, että widgetit muodostavat hierarkian, joka perustuu kokoonpanojärjestykseen. Jokainen widgetti perii ominaisuudet ylemmältä widgetiltä. Esimerkiksi usein käytetty Container-widgetti on tehty useammasta widgetistä, jotka ovat vastuussa widgettien sijoituksista, väristä ja koosta. Container-widgetti sisältää LimitedBox-, ConstrainedBox-, Align-, Padding-, DecoratedBox- ja Transform-widgetit. Kehittäjä voi vaihtaa edellä mainittua hierarkiaa esimerkiksi sovelluksen käyttäjän tehdessä eleen, jolloin hierarkiassa voidaan widgettien paikkoja vaihtaa. (flutter.dev b.) Hyvänä esimerkkinä on sovelluksen navigaatio-widgetti. Navigaatiossa korkeimman tason widgettinä pidetään esimerkiksi Home-näkymää, joka voi sisältää useamman alemman tason widgetin. Käyttäjän painaessa navigaatiolinkkiä (esim. Profile) käsketään ohjelmistokehystä korvaamaan edellinen widgetti (Home) uudella widgetillä (Profile). Tämän jälkeen ohjelmistokehys vertaa uutta- ja vanhaa widgettiä ja päivittää käyttöliittymän näkymää. (flutter.dev b.)

4 Flutter/Dart verrattuna React Native/JavaScriptiin

Vaikkakin Flutter ja React Native edustavat järjestelmäriippumattomia ohjelmistokehyksiä, on näiden kahden teknologian välillä eroja. Eroavaisuudet eivät pelkästään jää teknologian kehittäjään ja ohjelmointikieleen, jota teknologiat käyttävät, vaan niitä löytyy lisää UI-komponenteista, suorituskyvyistä, suosiosta ja monista muista asioista. Tässä luvussa käymme läpi muutamia avaineroavaisuuksia.

4.1 Ohjelmointikieli

React Native käyttää JavaScriptiä järjestelmäriippumattomien sovelluksien rakentamiseen. JavaScript on neljänneksi suosituin ohjelmointikieli vuonna 2019 (Putano 2019). Sitä käytetään myös Facebookin kehittämässä erittäin suositussa web-sovelluskehyksessä nimeltään React ja myös muissa suosituissa JavaScript-ohjelmistokehyksissä. JavaScript on dynaaminen ohjelmointikieli, joka mahdollistaa kehittäjän suorittaa melkein mitä tahansa tällä kielellä. Tämä seikka voi olla saamaan aikaan niin hyvä kuin huonokin asia. (Shashikant.)

Flutter käyttää sovelluksien rakentamiseen Dart-ohjelmointikieltä, jonka Google kehitti. Dart ei ole saanut vielä suurta käyttäjäryhmää, ja näin ollen se ei näy monissa suosituimmissa ohjelmointikielivertailuissa. Ohjelmointikieli on kuitenkin helppo omaksua sen ollessa hyvin samankaltainen suosituimpien ohjelmointikielien kanssa.

4.2 UI-komponentit

React Native -ohjelmistokehityspaketti tarjoaa kehittäjälleen vakiona ainoastaan UI-renderöintiin tarvittavat peruskomponentit eli näkymät, tekstit, tekstinsyöttökentät, napit, valinnat, tyylittelyt ja niin edespäin. Näiden lisäksi laitehallinta kuuluu pakettiin. Paketti ei kuitenkaan sisällä esimerkiksi navigointia, vaan tämä joudutaan asentamaan kolmannen osapuolen kirjastona ja usein näilläkin kirjastoilla on omia riippuvaisuuksia. Toki React Nativelle on tarjolla useampia versiota navigoinnista, ja tämä puute tavallaan antaa kehittäjälle vapauden valita itselleen sopivimman vaihtoehdon.

Flutter-ohjelmistokehityspaketti vuorostaan tarjoaa kehittäjälle todella paljon valmiita työkaluja sovelluksen rakentamiseen. Esimerkiksi Flutter sisältää valmiiksi samat UI-renderöintikomponentit kuin React Native ja jopa enemmänkin: laitehallinnan, navigaatiot ja testauksen. Flutter-navigaatio sisältää jo tarvittavat kirjastot käyttäjän navigointieleisiin ja navigointielementtien animoimiseen, eikä näin ollen vaadi useamman kirjaston asentamista projektia varten.

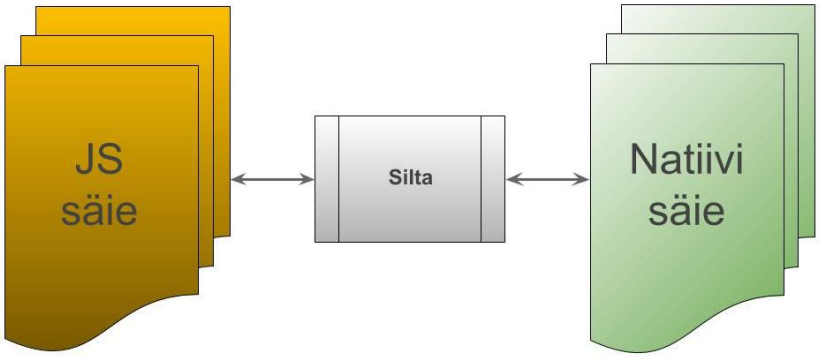
4.3 Suosio

React Native on yksi suosituimpia järjestelmäriippumattomia ohjelmistokehyksiä tällä hetkellä (Manchanda 2019). React Native pystyy suoriutumaan helposti niin suuresta kuin pienestä projektista. React Nativea käyttävät esimerkiksi Facebook, Instagram, Airbnb, Skype ja monet muut. (Ratnottar 2019.) React Nativella on myös suuri kehittäjäyhteisö GitHubissa ja React Nativesta järjestetään tapaamisia sekä konferensseja (Shashikant).

Flutter on vielä suhteellisen uusi tulokas järjestelmäriippumattomissa ohjelmistokehyksissä ja yrittää vieläkin saada jalansijaa markkinoilla. Flutterin käyttäjiä ovat Alibaba, Tencent, Abbey Road Studios ja Google. (flutter dev c.) Flutterin kannatus on kuitenkin ajan myötä vain kasvanut (Google Trends) ja tapaamisia sekä konferenssejä on järjestetty verkossa jo paljon. (Shashikant.)

4.4 Koonti ja natiivisuus

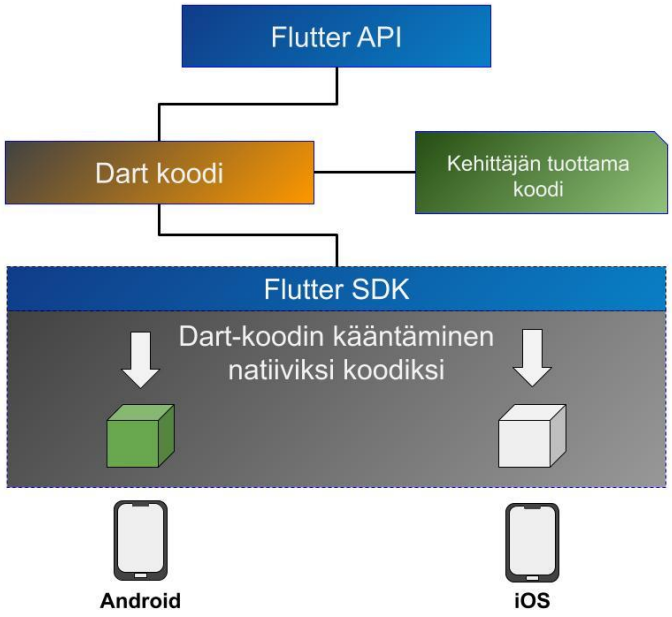
React Nativen koodin koontivaiheessa koodia ei muuteta natiiviksi koodiksi, vaan React Nativen JavaScript-koodi pystyy keskustelemaan natiivin koodin kanssa käyttämällä ”siltaa” näiden välillä. Tämä silta on konsepti, joka mahdollistaa kaksisuuntaisen ja asynkronisen keskustelun koodien välillä (ks. kuva 5). Koodien välinen silta on rakennettu käyttämällä C- ja C++-ohjelmointikieliä, mikä mahdollistaa koodin ajamisen useammalla käyttöjärjestelmällä.



Kuva 5. Kuvassa havainnollistetaan, miten JavaScript-säie antaa asynkronisesti ja kaksisuuntaisesti käskyjä natiiville säikeelle.

Esimerkkinä silta konseptin mahdollisesta toiminnasta on <View>-elementti. JavaScript lähettää sillan kautta tiedot näkymästä natiiville puolelle, jonka natiivi koodi joutuu renderöimään. Tällä periaatteella kaikki UI-tapahtumat ja renderöintikäskyt liikkuvat. (Frachet 2019.)

Flutter käyttää koodin koontivaiheessa omaa moottoriaan ja muuntaa ohjelmistokehityspaketin ja kehittäjän Dart-koodin natiiviksi koodiksi. Android-käyttöjärjestelmille käännettäessä koodi käännetään natiiviksi ARM- ja x86-kirjastoiksi. iOS-käyttöjärjestelmiä varten koodi käännetään käyttäen apuna LLVM-kääntäjää, jolla Dart-koodi muutetaan natiiviksi ARM-kirjastoksi (ks. kuva 6). (flutter.dev.faq a; flutter.dev.faq b.)



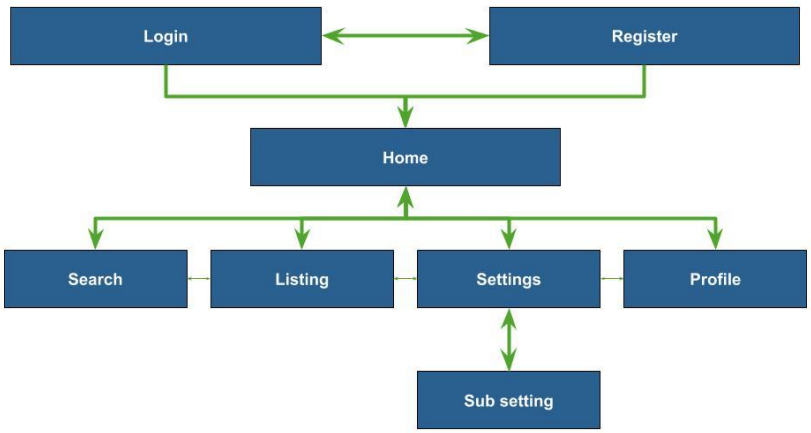
Kuva 6. Havainnollistaminen miten Flutter kääntää Dart-koodin natiiviksi koodiksi.

Periaatetasolla kehittäjän tuottama Dart-koodi ei ole sama kuin valitulla käyttöjärjestelmällä ajettava koodi. Käännösvaiheen jälkeen ajettaessa koodia tuotetut kirjastot sisällytetään käyttöjärjestelmissä runner-projektiin, jonka jälkeen tämä runner-projekti kootaan Android-käyttökäyttöjärjestelmille APK:ksi ja iOS-käyttöjärjestelmille .ipa:ksi. Kaikki renderöinti, käyttötapahtumat ja syötteet delegoidaan suoraan käännetylle Flutter- ja sovelluskoodille. Tätä mallia monet pelimoottoritkin käyttävät (flutter.dev.faq a).

5 Testiohjelmistot

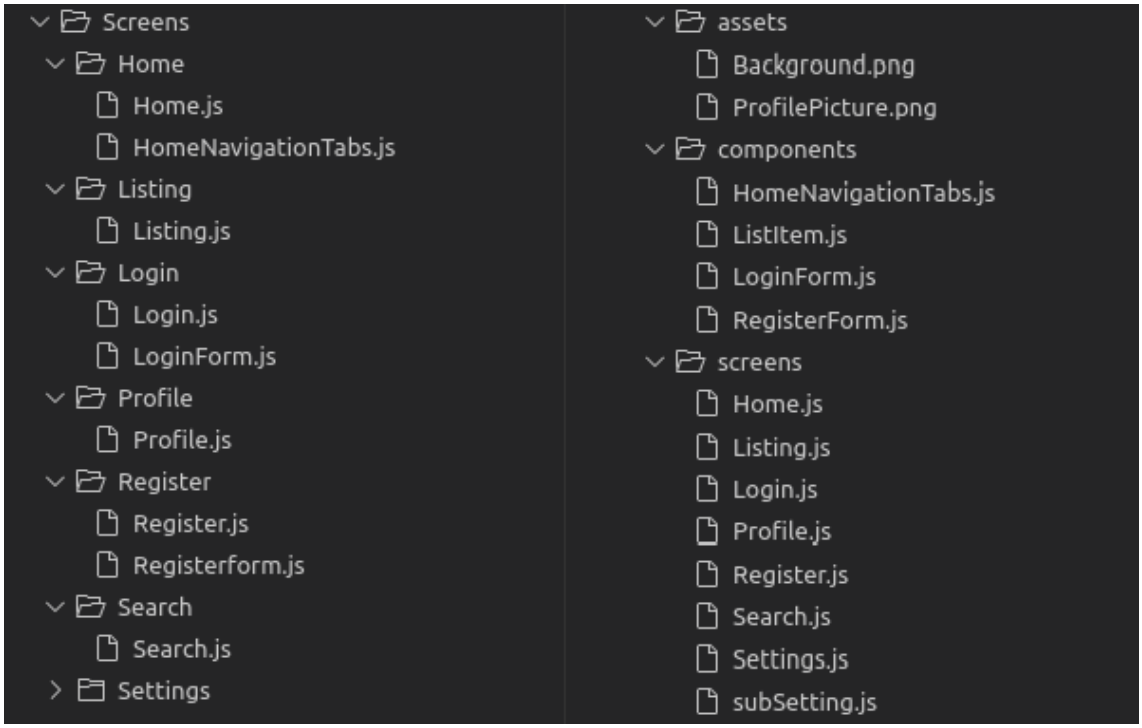
Seuraavaksi insinöörityössä tehdään molemmista edellä esitetyistä järjestelmäriippumattomista ohjelmistokehysten avulla sovellukset. Sovellukset tulevat keskittymään vain käyttöliittymään liittyviin elementteihin ja näkymiin. Vaikkakin molemmat teknologiat tarjoavat datanhakutyökalut jo valmiiksi, on tämän insinöörityön tarkoituksena vain tarkastella näitä teknologioita pelkästään UI-kehittäjän näkökulmasta. Ohjelmat tullaan kehittämään käyttämällä teknologioiden tarjoamia dokumentteja. Samalla mahdollistetaan näiden vertailu.

Käyttöliittymien suunnittelu aloitetaan piirtämällä navigointikartta, jossa määritellään ylä- ja alatason näkymät. Ylätasoihin sisältyy Login-, Register-, Home-, Search-, Listing-, Settings- ja Profile-näkymä. Alatasoihin luetteltaisiin kaikki näkymät, jotka olisivat ylätason alla. Jotta käyttöliittymissä olisi mahdollisuus siirtyä muihinkin näkymiin, tarvitaan ohjelmistoon navigaatio. Navigointiin valitaan päänavigaatioelementiksi vetolaatikkonavigaatio (drawer navigation) sekä tarvittaessa sovelluksen yläosaan lisättävä navigaatiopalkki (top navigation bar). Vetolaatikkonavigaatio on kätevä tapa navigoida mobiilikäyttöliittymissä. Se liukuu yleensä käyttäjän toimesta sovelluksen vasemmasta reunasta tuoden samalla esiin navigaatiomahdollisuudet useampaan näkymään. Lisäksi vetolaatikkotyylinen navigointi antaa enemmän tilaa luoda sisältöä sovelluksen näkymiin. Sovelluksen navigaatiopalkki antaa perinteisesti käyttäjälle tiedon esillä olevasta näkymästä lisäämällä näkymän nimen palkkiin. Käyttöliittymien näkymissä navigoiminen tulee olemaan yksi- ja monisuuntaista. Ylätason näkymistä voidaan siirtyä mihin tahansa toiseen ylätason näkymään (poikkeuksena Login ja Register, Home-näkymään siirtymisen jälkeen) ja vain eteen ja taaksepäin alemman tason näkymistä (SubSetting). (ks. kuva 7.)



Kuva 7. Testiohjelmistojen navigaatio rakennetaan kuvan esittämällä tavalla. Siniset laatikot edustavat ohjelmistojen näkymiä ja vihreät nuolet navigaatiomahdollisuuksia.

Navigaatiokartan luomisen jälkeen on tärkeää päättää, miten projektin tiedostorakenne tullaan määrittämään. Tämä helpottaa projektin kasvaessa oikeiden komponenttien ja widgettien löytämistä. Yleisimmät tavat ovat luokitella käyttöliittymän näkymät ja siihen kuuluvat komponentit omiin kansioihin tai jakaa tiedostot niiden tyypin mukaan eli komponentit ja näkymät omiin kansioihin (ks. kuva. 8). Tähän insinöörityöhön valitaan tiedoston tyypin mukainen lajittelu.



Kuva 8. Tiedostojen lajittelu kahdella tavalla. Vasemmalla puolella tiedostot lajitellaan näkymien mukaan ja oikealla lajitellaan tiedoston tyypin mukaan.

Kun tiedostorakenne on valittu ja navigaatiokartta on luotu, voidaan aloittaa käyttöliittymien kehitys. Käyttöliittymien kehitys aloitetaan luomalla valittu tiedostorakenne, jossa lisätään ylemmän tason näkymät. Lisäksi muutan molempien käyttöliittymien juuritiedoston navigaatiota varten (React Nativen App.js- ja Flutterin main.dart-tiedosto). Flutterin sisältäessä jo navigaation joudun valitsemaan navigaatiokirjaston React Nativelle. Valittu kirjasto tähän työhön on React Navigation.

5.1 Käyttöliittymien navigointi

React Nativella luodaan juuritiedostoon AppContainer-nimisen komponentin, joka luodaan React Navigation -kirjaston tarjoamalla createApplicationContainer-funktiolla. AppContainer renderöidään juuritiedostossa. AppContaineriin lisäämme AppSwitchNavigator-komponentin, joka mahdollistaa liikkumisen kirjautumis- ja rekistöitymisnäkymiin ja niistä Home-näkymään. Switch-navigaation jälkeen lisäämme Home-näkymään vielä vetolaatikkonavigaation ja sen sisään Stack-navigaation. Lopulta määrittelemme HomeStack-näkymäksi HomeScreen-näkymän (ks. kuva 9). Tällä navigaatiorakenteella voimme luoda kaikki mahdolliset näkymät ja navigoida käyttöliittymän näkymissä. Navigaation täysiversio löytyy liitteestä yksi.

```
// Home screen StackNavigator which returns the actual Home screen
const HomeStack = createStackNavigator ({
  Home: {
    screen: HomeScreen,
    navigationOptions: ({ navigation }) => {
      return {
        headerTitle: 'Home',
        headerLeft: (<Icon style={{ paddingLeft: 10 }} onPress={() => navigation.openDrawer()} name="md-menu" size={30} />)
      }
    }
  },
});

// Drawer navigation for moving between all upper level screens
// All screens have stack as a screen because the screen will be defined in StackNavigator
const AppDrawerNavigator = createDrawerNavigator ({
  Home:{
    screen: HomeStack,
  },
  Search:{
    screen: SearchStack,
  },
  Listing:{
    screen: ListingStack,
  },
  Settings:{
    screen: SettingsStack,
  },
  Profile:{
    screen: ProfileStack,
  },
},
{
  drawerPosition: 'left',
  drawerBackgroundColor: '#fafafa',
  drawerType: 'slide',
}
);

// Navigation component for moving between Login <-> Register and to Home
// Home has screen AppDrawerNavigation because the screen will be defined in StackNavigator
const AppSwitchNavigator = createSwitchNavigator({
  LogIn: { screen: LoginScreen },
  Register: { screen: RegisterScreen },
  Home: { screen: AppDrawerNavigator }
});

// Create a container for application navigation and render it as a root
const AppContainer = createAppContainer(AppSwitchNavigator)
```

Kuva 9. Esimerkki React Nativen navigoinnista.

Flutterilla navigoimisen rakentaminen on paljon yksinkertaisempaa kuin React Nativella. Koska navigointi on sisäänrakennettu Flutteriin, voimme käyttää sitä välittömästi. Navigoiminen Flutterilla tapahtuu määrittelemällä ensimmäinen widgetti, joka halutaan renderöidä. Tämän jälkeen lisätään polkulistaus. Poluille annetaan listauksessa string-tunniste ja sille sisältö, joka on tässä tapauksessa tuodut näkymät main.dart-tiedostoon. Sisällöt renderöidään WidgetBuilderilla. Kun käyttäjä tekee navigointipyynnön, ohjelma tarkastaa reittimäärityksistä kaikki määritetyt reitit string-tunnisteen mukaan. Löytäessään täsmäävän tunnisteen näkymä renderöidään (ks. kuva 10).

```
// Screen imports
import 'screens/Login.dart';
import 'screens/Register.dart';
import 'screens/DashBoard.dart';
import 'screens/Search.dart';
import 'screens/Listing.dart';
import 'screens/Settings.dart';
import 'screens/Profile.dart';

// Start application MyApp
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  // Routes for all imported widgets
  final routes = <String, WidgetBuilder> {
    'register': (context) => Register(),
    'dashboard': (context) => DashBoard(),
    'search': (context) => Search(),
    'listings': (context) => Listing(),
    'settings': (context) => Settings(),
    'profile': (context) => Profile(),
  };

  @override
  Widget build(BuildContext context) {
    return MaterialApp (
      title: 'Flutter Frame',
      debugShowMaterialGrid: false,
      theme: ThemeData(
        primarySwatch: Colors.teal,
        fontFamily: 'Montserrat'
      ), // ThemeData
      home: Login(), // First widget to be rendered after startup
      routes: routes, // When navigation is called App will seek for matching navigation string
    ); // MaterialApp
  }
}
```

Kuva 10. Flutter-sovelluksen main.dart-tiedosto, jossa on määritelty jokaisen näkymän polku.

Jotta navigoiminen onnistuisi ja Flutter löytäisi oikean näkymän navigointitilanteessa, on äärimmäisen tärkeää merkitä jokainen näkymä samalla string-tunnuksella kuin reittimäärityksissä. Esimerkiksi register-näkymän tiedostoon täytyy merkitä seuraavanlainen koodirivi.

```
Static String tag = 'register';
```

Esimerkkikoodi 7. Register-näkymän string-tunnus navigaatiota varten. Koodirivi täytyy lisätä jokaiseen tiedostoon vastaamaan main.dart-tiedoston routes string -määrityksiä.

5.2 Käyttöliittymien kirjautuminen sekä rekisteröityminen

Molemmat käyttöliittymät tulevat alkamaan suorittamalla kirjautumis- tai rekisteröitymisnäkymä. Näkymissä tulee olemaan tekstinsyöttökenttiä käyttäjän tiedoille, sekä kaksi navigoimiseen tarkoitettua elementtiä. Itse ohjelmaan siirtyminen tulee tapahtumaan käyttäjän painaessa Sign In- tai Register-nappia (ks. kuva 11).

Kuva 11. Kirjaudu- sekä rekisteröidy-näkymät. Molemmissa näkymissä ohjelmistot ottavat vastaan käyttäjän syöteitä tekstikentissä.

React Nativella hajautetaan kirjautumis- ja rekisteröintilomakkeet omiin komponenttitiedostoihin (ks. kuva 8). Tekstikentät (TextInput) tulevat keräämään käyttäjän syötteen ja tallentamaan sen jokaiselle tekstikentälle uniikkiin tilaan (ks. kuva 12). (this.state.)

```
class LoginForm extends React.Component {
  constructor(props) {
    super(props);
    // States for all possible TextInput elements
    this.state = {
      username: "",
      usernameError: "",
      password: "",
      passwordError: ""
    };
  }

  render() {
    return (
      <View style={styles.container}>
        <TextInput
          style={styles.inputBox}
          underlineColorAndroid="rgba(0, 0, 0, 0)"
          placeholderTextColor='#333333'
          name="username"
          placeholder='Username'
          autoCapitalize="none"
          autoCorrect={false}
          autoFocus={false}
          keyboardType="email-address"
          value={this.state.username}
          // Call this whenever user inputs something to update username state
          onChangeText={text => this.setState({ username: text })}
        />
```

Kuva 12. React Native -tekstinsyöttökenttä ja tilat.

Molemmille lomakkeille tehdään samankaltaiset tiedostot. Lomakkeiden eroavaisuudet ovat ainoastaan syötekenttien (TextInput) määrä ja niiden sisältö. Jotta Login- ja Register-näkymät toimisivat oikein, on aika lisätä juuri luodut komponentit niille kuuluviin näkymiin. Esimerkkinä Login-näkymän tiedosto (ks. kuva 13).

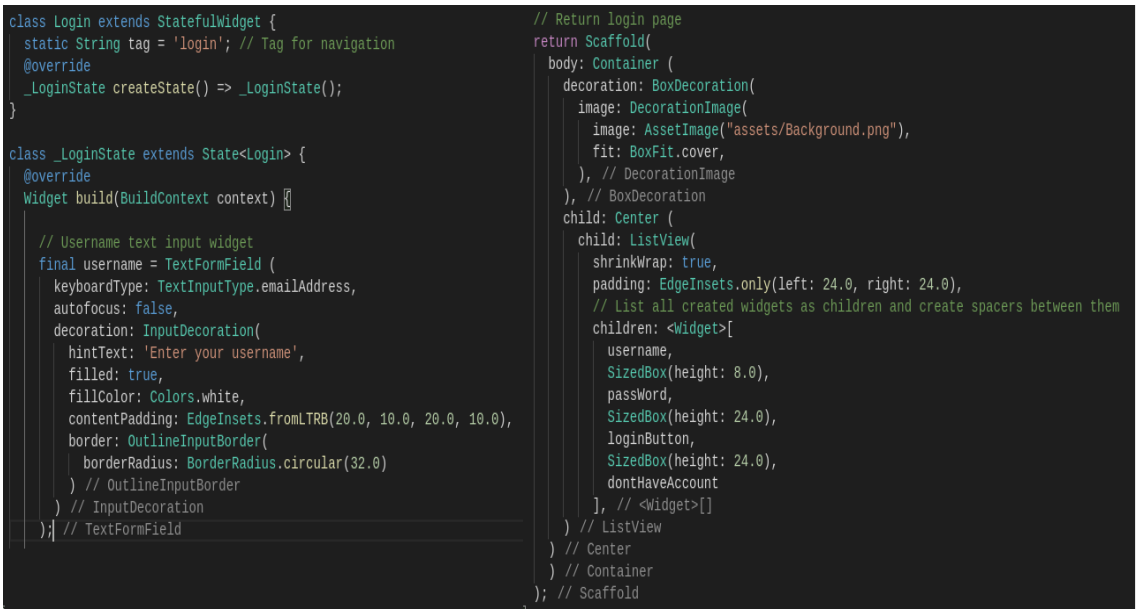
```
// Import Login form from components folder
import LoginForm from '../components/LoginForm';
import Background from '../assets/Background.png';

export default class Login extends Component {
  render() {
    return (
      <ImageBackground
      source={Background}
      style={styles.container}
      >
        {/* Render login form */}
        <LoginForm/>
        <View style={styles.registerTextCont}>
          <Text style={styles.generalText}>Don't have an Account?</Text>
          {/* Navigate to Register screen */}
          <TouchableOpacity onPress={() => this.props.navigation.navigate('Register')}>
            <Text style={styles.registerText}> Sign Up</Text>
          </TouchableOpacity>
        </View>
      </ImageBackground>
    );
  }
};
```

Kuva 13. Login.js-tiedosto, jossa tuodaan itse luotu LoginForm-komponentti ja renderöidään se Login-luokassa.

Jotta navigoiminen olisi mahdollista näiden kahden näkymän välillä, on molempiin näkymiin lisätty elementti TouchableOpacity, jota käyttäjän painaessa navigaatio vaihtaa näkymää Login- ja Register-näkymän välillä (ks. kuva 13). Täydelliset React Native Login- ja Register-näkymät sekä niiden toiminnot ovat liitteessä yksi.

Flutterin kirjautumis- ja rekisteröitymisnäkymät ovat rakenteeltaan hyvin samanlaisia kuin React Nativella. Näkymiin määritellään widgettejä (ks. kuva 14), jotka liitetään myöhemmin näkymien renderöintiin (ks. kuva 14).



```
class Login extends StatefulWidget {
  static String tag = 'login'; // Tag for navigation
  @override
  _LoginState createState() => _LoginState();
}

class _LoginState extends State<Login> {
  @override
  Widget build(BuildContext context) {

    // Username text input widget
    final username = TextFormField (
      keyboardType: TextInputType.emailAddress,
      autofocus: false,
      decoration: InputDecoration(
        hintText: 'Enter your username',
        filled: true,
        fillColor: Colors.white,
        contentPadding: EdgeInsets.fromLTRB(20.0, 10.0, 20.0, 10.0),
        border: OutlineInputBorder(
          borderRadius: BorderRadius.circular(32.0)
        ) // OutlineInputBorder
      ) // InputDecoration
    ); // TextFormField
```

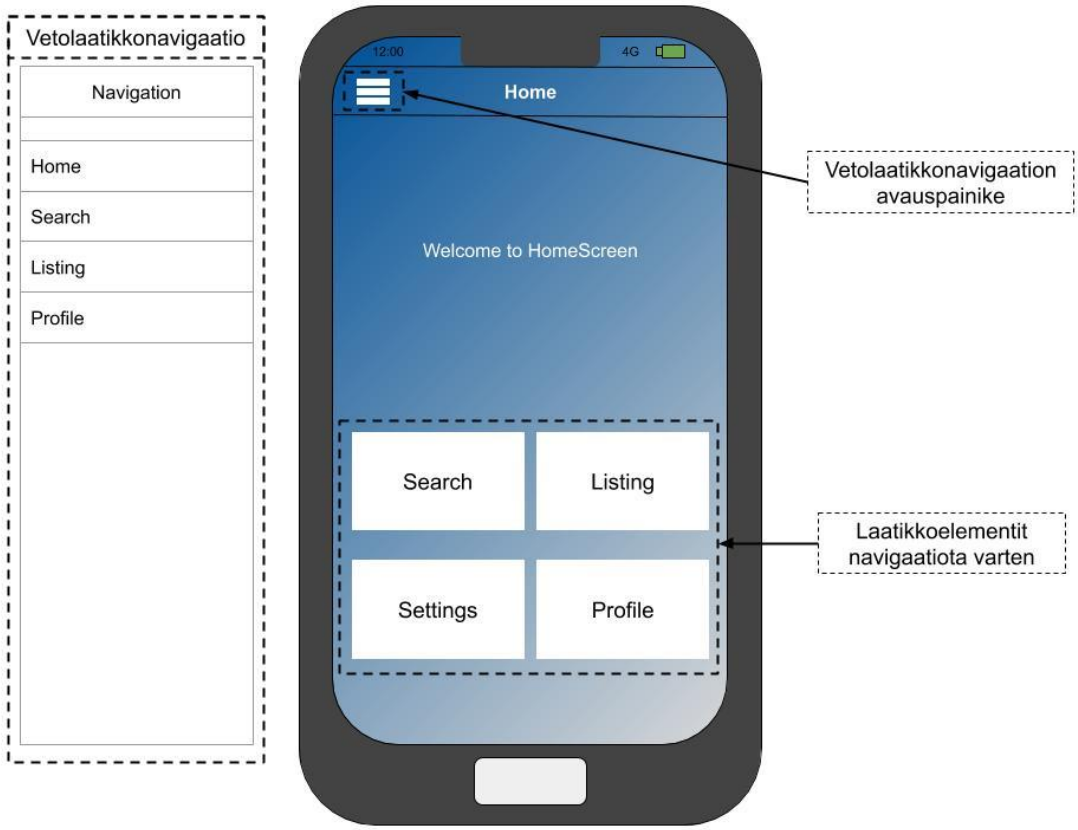
```
// Return login page
return Scaffold(
  body: Container (
    decoration: BoxDecoration(
      image: DecorationImage(
        image: AssetImage("assets/Background.png"),
        fit: BoxFit.cover,
      ), // DecorationImage
    ), // BoxDecoration
    child: Center (
      child: ListView(
        shrinkWrap: true,
        padding: EdgeInsets.only(left: 24.0, right: 24.0),
        // List all created widgets as children and create spacers between them
        children: <Widget>[
          username,
          SizedBox(height: 8.0),
          passWord,
          SizedBox(height: 24.0),
          loginButton,
          SizedBox(height: 24.0),
          dontHaveAccount
        ], // <Widget>[]
      ) // ListView
    ) // Center
  ) // Container
); // Scaffold
```

Kuva 14. Vasemmalla esitetään, miten Flutterilla luodaan tekstinsyöttökenttä widgetti nimeltä username. Oikealla esitetään, miten kyseinen widgetti lisätään Login-näkymään.

Widgettien luominen ei ole rajoitettu pelkästään kuvan 14 esittämällä tavalla, vaan widgetit voidaan myös määritellä suoraan return-lausekkeessa. Erikseen määriteltynä koodi on kuitenkin huomattavasti helpompaa lukea ja ylläpitää. Jokaiseen näkymään on määritettävä navigaation tunniste (ks. kuva 14) samalla tavalla kuin esimerkkikoodi seitsemässä on määritetty. Flutter-käyttöliittymän Login- ja Register-näkymien koodit ja toiminnot löytyvät liitteestä kaksi.

5.3 Kotinäkymä

Sovelluksien keskukseksi luodaan kotinäkymä, jossa käyttäjä voi valita seuraavat toimensa. Kotinäkymään sijoitetaan laatikkoelementtejä, joista voidaan navigoida seuraaviin näkymiin. Laatikkojen lisäksi näkymä tulee sisältämään navigaatiopalkin näkymän yläosassa, joka kertoo käyttäjälle missä näkymässä ollaan. Navigaatiopalkissa on myös vasemmassa reunassa navigaatioikoni, josta painalla sivulta avautuu laatikkonavigaatio (ks. kuva 15).



Kuva 15. Kotinäkymä, jossa kaksi erilaista navigaatiomahdollisuutta.

React Nativella luodaan kotinäkymän navigaatiolaatikot erillisenä komponenttina. Komponenttia luodessa voi määritellä listan objekteja, joista jokainen objekti edustaa edellä mainittuja laatikoita (ks. kuva 16). Lisäksi tarvittavat elementit voidaan helposti luoda JavaScriptin map-funktiolla (ks. kuva 16). Map-funktio käy NavTabs-listan läpi yksi objekti kerrallaan ja palauttaa jokaista objektia kohden TouchableOpacity-elementin.

```
// All Home screen navtabs data in array
const navTabs = [
  {key: 0, name: 'Search', navLink: 'Search'},
  {key: 1, name: 'Listing', navLink: 'Listing'},
  {key: 2, name: 'Settings', navLink: 'Settings'},
  {key: 3, name: 'Profile', navLink: 'Profile'}
];

class HomeNavigationTabs extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        {/*
          Start mapping navTabs array and create
          a touchableOpacity for each object in array
          with item data
        */}
        {navTabs.map((item) => {
          return(
            <TouchableOpacity
              key={item.key}
              style={styles.navTab}
              onPress={() => this.props.navigation.navigate(`${item.navLink}`)}>
              <Text style={styles.navText}>{item.name}</Text>
            </TouchableOpacity>
        )})}
      </View>
    );
  }
}

export default withNavigation(HomeNavigationTabs);

// Styles for Navigation Tabs
const styles = StyleSheet.create({
  container: {
    justifyContent: "center",
    alignItems: "center",
    // Row rule to get items side by side in map() function
    flexDirection: 'row',
    flexWrap: 'wrap',
```

Kuva 16. Kotinäkymän navigaationlaatikoiden koodi. Koodissa määritellään navTabs-lista, jonka tiedoilla luodaan elementtejä renderöintiosiossa. Elementit luodaan käyttämällä JavaScriptin-funktiota map().

Jotta näkymässä saataisiin navigaatiot ruudukkoon kuvan 15 osoittamalla tavalla, on lisättävä tyylittelyyn flexDirection-sääntö ja sen arvoksi ”row”. Tämä sääntö pakottaa käyttöliittymän renderöimään elementit sivuttaissunnassa. Ilman tätä sääntöä elementit renderöitäisiin allekkain. Tämän jälkeen HomeNavigationTabs-komponentti tuodaan ja renderöidään Home.js-tiedostossa (ks. kuva 17). Täydellinen koodi löytyy liitteestä yksi.

```
import HomeNavTabs from '../components/HomeNavigationTabs';
import Background from '../assets/Background.png';

export default class Home extends Component {
  render() {
    return (
      <ImageBackground
      source={Background}
      style={styles.container}
      >
      <Text style={styles.welcomeText}>Welcome to HomeScreen</Text>
      <HomeNavTabs style={styles.navigationTabs} />
      </ImageBackground>
    );
  }
};
```

Kuva 17. Home.js-tiedosto, jossa renderöidään HomeTabs-navigaatiokomponentti sekä teksti elementti (Text).

Flutter-kotinäkymä rakennetaan samalla periaatteella kuin Login- ja Register-näkymät. Navigaatiolaatikoista tehdään kaksi erillistä widgettiä. Widgetit navBoxesFirst ja NavBoxesSecond sisältävät kaksi MaterialButton-widgettiä navigointitietoineen (ks. kuva 18).

```
final navBoxesFirst = Row (
  mainAxisAlignment: MainAxisAlignment.center,
  crossAxisAlignment: CrossAxisAlignment.center,
  children: <Widget>[
    Padding (
      padding: EdgeInsets.symmetric(vertical: 20.0),
      child: Material (
        borderRadius: BorderRadius.circular(30),
        shadowColor: Colors.black38,
        elevation: 5.0,
        child: MaterialButton(
          minWidth: 260.0,
          height: 150.0,
          // When box is pressed navigate normally to search screen
          onPressed: () {
            Navigator.pushNamed(context, 'search');
          },
          color: Colors.white,
          child: Text('Search', style: TextStyle(color: Colors.black)),
        ), // MaterialButton
      ), // Material
    ), // Padding
    Padding (
      padding: EdgeInsets.symmetric(vertical: 20.0, horizontal: 20.0),
      child: Material (
        borderRadius: BorderRadius.circular(30),
        shadowColor: Colors.black38,
        elevation: 5.0,
        child: MaterialButton(
          minWidth: 260.0,
          height: 150.0,
          // When box is pressed navigate normally to listing screen
          onPressed: () {
            Navigator.pushNamed(context, 'listings');
          },
          color: Colors.white,
          child: Text('Listing', style: TextStyle(color: Colors.black)),
        ), // MaterialButton
      ), // Material
    ) // Padding
  ], // <Widget>[]
); // Row
```

Kuva 18. Navigointi-widgetit.

Navigointi-widgetit lisätään renderöintiin samalla tavalla kuin Login- ja Register-näkymien tekstikentät (ks. kuva 14).

```dart
// Add Drawer navigation to dashboard/home screen
drawer: Drawer(
  child: ListView(
    padding: EdgeInsets.zero,
    children: <Widget>[
      DrawerHeader(
        child: Text(''),
        decoration: BoxDecoration(
          color: Colors.teal,
        ), // BoxDecoration
      ), // DrawerHeader
      // List all possible navigation links to drawer
      ListTile(
        title: Text('Home'),
        // onTap to navigate normally to desired screen
        onTap: () {
          Navigator.pushNamed(context, 'dashboard');
        },
      ), // ListTile
      ListTile(
        title: Text('Search'),
        // onTap to navigate normally to desired screen
        onTap: () {
          Navigator.pushNamed(context, 'search');
        },
      ), // ListTile
      ListTile(
        title: Text('Listing'),
        // onTap to navigate normally to desired screen
        onTap: () {
          Navigator.pushNamed(context, 'listings');
        },
      ), // ListTile
      ListTile(
        title: Text('Settings'),
        // onTap to navigate normally to desired screen
        onTap: () {
          Navigator.pushNamed(context, 'settings');
        },
      ), // ListTile
      ListTile(
        title: Text('Profile'),
        // onTap to navigate normally to desired screen
        onTap: () {
          Navigator.pushNamed(context, 'profile');
        },
      ), // ListTile
    ], // <Widget>[]
  ), // ListView
```

```dart
// return dashboard/home screen
return Scaffold(
  // Add topnav bar to screen
  appBar: AppBar(
    centerTitle: true,
    title: Text('Home'),
  ), // AppBar
  body: Container (
    decoration: BoxDecoration(
      image: DecorationImage(
        image: AssetImage("assets/Background.png"),
        fit: BoxFit.cover,
      ), // DecorationImage
    ), // BoxDecoration
    child: Column (
      mainAxisAlignment: MainAxisAlignment.center,
      crossAxisAlignment: CrossAxisAlignment.center,
      children: <Widget>[
        welcomeText,
        SizedBox(height: 60.0),
        navBoxesFirst,
        navBoxesSecond,
      ], // <Widget>[]
    ), // Column
  ), // Container
```

Kuva 19. Vasemmalla esitetään Flutterin vetolaatikkonavigaation lisääminen näkymään. Oikealla esitetään navigaation yläpalkin lisääminen näkymään.

Lisäksi joudumme aina lisäämään jokaiseen näkymään vetolaatikkonavigaation (Drawer) ja yläpalkin (AppBar), jotta käyttöliittymässä pystytään navigoimaan näkymien kesken (ks. kuva 19). Vetolaatikon (Drawer) reitit määritetään lisäämällä vetolaatikkoon alatason widgeteiksi ListTile-widgettejä ja näille onTap-funktio. Täydellinen koodi löytyy liitteestä kaksi.

5.4 Hakunäkymä

Hakunäkymä ei tule sisältämään muita elementtejä kuin hakukentän ja vetolaatikkonavigaation. Haku ei tule suodattamaan mitään tietoa näkymästä eikä hae tietoja mistään taustajärjestelmistä. (ks. kuva 20.)

Kuva 20. Yksinkertainen hakunäkymä, jossa on vain yksi syötekenttä.

React Nativella hakukentän luominen suoritetaan samalla periaatteella kuin Login- ja Register-näkymien tekstisyötekentät (ks. kuva 12). Tiloihin (state) määritellään vain hakuun sopiva nimitys esimerkiksi ”value” tai ”searchValue”. Määritetty uusi tila vaihdetaan syötekentän value- ja onChangeText-arvoihin (ks. kuva 12). Täydellinen koodi löytyy liitteestä yksi.

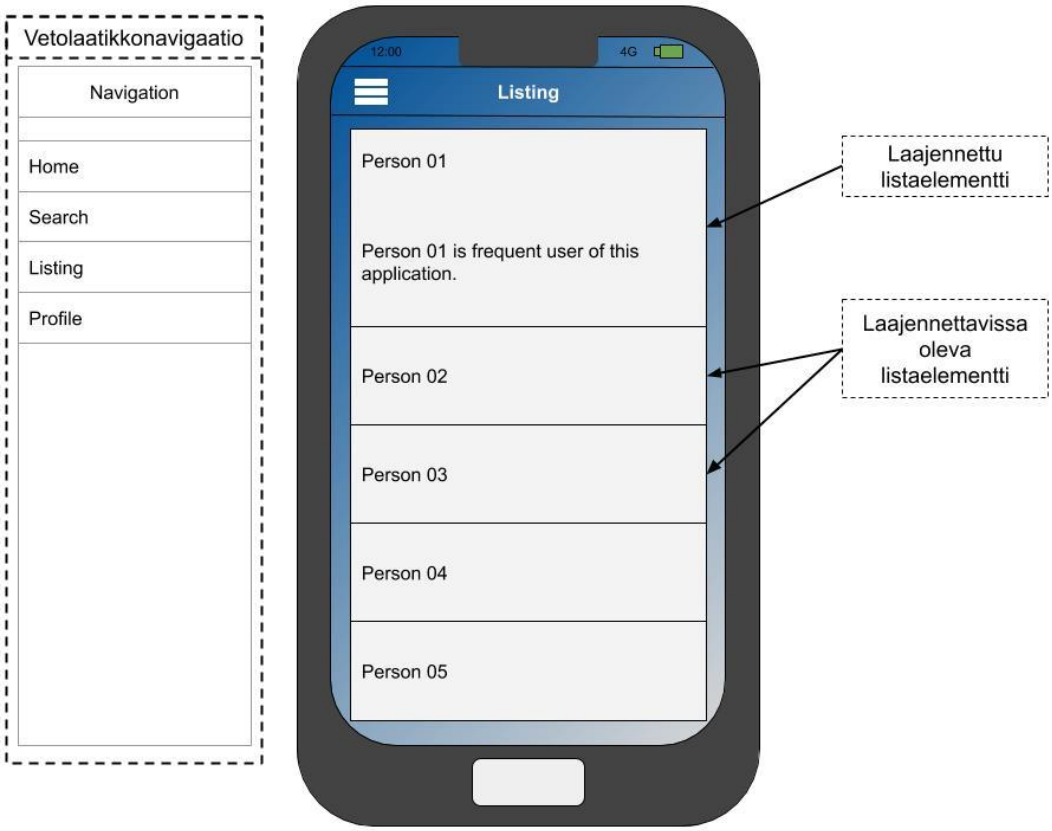
```
class _SearchState extends State<Search> {
  // Screen title as widget because it will be hidden when doing search
  Widget appBarTitle = Text('Search');
  // Define search icon. Icon will be changed after state change
  Icon openSearch = Icon(Icons.search, color: Colors.white);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
       centerTitle: true,
       title: appBarTitle,
       actions: <Widget>[
         IconButton(
           icon: openSearch,
           // When user clicks on search icon ->
           // change appbar title to textform widget and change icon to close icon
           // After user clicks on close icon remove textform widget and display screen title
           onPressed: () {
             setState(() {
               if (this.openSearch.icon == Icons.search) {
                 this.openSearch = Icon(Icons.close);
                 this.appBarTitle = TextFormField(
                   decoration: InputDecoration(
                   hintText: 'Search for something',
                   filled: true,
                   fillColor: Colors.white,
                   contentPadding: EdgeInsets.fromLTRB(20.0, 10.0, 20.0, 10.0),
                   border: OutlineInputBorder(
                     borderRadius: BorderRadius.circular(32.0)
                   ) // OutlineInputBorder
                   ) // InputDecoration
                 ); // TextFormField
               } else {
                 this.openSearch = Icon(Icons.search);
                 this.appBarTitle = Text('Search');
               }
             });
           },
         ) // IconButton
       ], // <Widget>[]
      ), // AppBar
      // Add drawer navigation to screen
       drawer: Drawer(
```

Kuva 21. Flutterin hakukentän sisällyttäminen navigaatiopalkkiin.

Flutterilla hakukenttä sisällytetään näkymän navigaatiopalkkiin (AppBar). Navigaatiopalkkiin lisätään hakuikoni (Icon), jota painamalla hakukenttä avautuu navigaatiopalkissa. Hakunäkymässä käytetään saman tyylistä tilan hallintaa kuin React Nativessa. Käyttäjän painaessa hakuikonia vaihdetaan määritetyn appBarTitle-widgetin tilalle textFormField-widgetti (ks. kuva 21). Täydellinen koodi löytyy liitteestä kaksi.

5.5 Listausnäkymä

Sovelluksen sisältämä listanäkymä (Listing) sisältää listan elementtejä, joita sovelluksen käyttäjä pystyy valitsemaan. Valittaessa listasta elementin valittu elementti laajenee (ks. kuva 22).



Kuva 22. Listanäkymä, jossa listataan ennalta määriteltyjä objekteja sekä niiden data. Jokainen listaelementti on laajennettavissa sekä supistettavissa.

Listanäkymää luotaessa React Native-ohjelmistokehyksellä käytetään natiivikomponentteja nimeltään FlatList. FlatList-komponentille annetaan samanlainen datalista kuin Home-näkymän navigaatiolaatikoille (datan sisältö listanäkymän mukainen) (ks. kuva 16 ja 22). FlatList-komponentille kerrotaan, mitä renderöidään listanäkymässä (renderItem) (ks. kuva 23).

```
export default class Listing extends Component {
  render() {
    return (
      <ImageBackground
      source={Background}
      style={styles.container}
      >
        <FlatList
          // Set data
          data={data}
          // Render imported listviewItem
          renderItem={({item}) => <ListViewItem item={item}/>}
          // Tells list to use the id's for the react keys instead of default key property
          keyExtractor={(index, _) => index + ''}
        />
      </ImageBackground>
    );
  }
};
```

Kuva 23. React Native Listing -luokka, jossa käytetään natiivikomponenttia FlatList.

Kun FlatList on määritelty, voidaan rakentaa itse listauksessa käytettävät elementit. Listanäkymän elementit hajautetaan omaan ListItem-komponenttitiedostoon, jos listauskomponenttia tarvitsisi jossakin vaiheessa työtä uudestaan. Koska suunnitelmassa on laajentaa listan elementtiä käyttäjän valitessa sellaisen, joudutaan komponenttiin lisäämään ehdollinen renderöinti. Tämä saavutetaan lisäämällä jokaiseen listan elementtiin tila (state) selected ja tälle tilan muutokselle käsittelijäfunktio handleClick. HandleClick-käsittelijäfunktio vaihtaa selected-tilan true- tai false-arvon välillä riippuen siitä, mikä listaelemetin selected-tilan arvo oli funktiota kutsuttaessa. (ks. kuva 24.)

```
class ListItem extends Component {
  constructor(props) {
    super(props);
    // State for each list element
    // Default not showing subText
    this.state = {
      selected: false,
    };
  };

  handleClick = () => {
    // Change state of selected on user press action
    this.setState(prevState => ({
      selected: !prevState.selected
    }));
  };

  // Function to render subtext on list
  renderSubText = () => {
    return (
    <View style={styles.subContainer}>
      <Text style={styles.subText}>{this.props.item.subText}</Text>
    </View>
    )
  }
```

Kuva 24. ListItem-tila ja tilan selected muutoksen käsittelijä handleClick. Funktio renderSubText palauttaa View- ja Text-elementin funktiota kutsuttaessa.

Näiden funktioiden avuilla voidaan suorittaa renderöinnissä ehdollisen renderöinnin, jokaiselle listaelementille (ks. kuva 25). Täydellinen koodi löytyy liitteestä yksi.

```
render() {
  const { selected } = this.state;
  return (
    <View style={styles.container}>
      <TouchableWithoutFeedback onPress={this.handleClick}>
        <View style={styles.titleContainer}>
          <Text style={styles.titleText}>{this.props.item.heading}</Text>
        </View>
      </TouchableWithoutFeedback>
      {/* Conditional render for subtext */}
      {selected && this.renderSubText()}
    </View>
  );
}
```

Kuva 25. ListItem-luokan renderöinti, sekä sen sisäinen ehdollinen renderöinti. Ehdollinen renderöinti aktivoituu vain käyttäjän painaessa listaelementtiä.

Rakennettaessa Flutterin versiota listauksesta käytetään siinä ”ListView.builderia”. Listview.builder toimii hyvin samalla tavalla kuin React Nativen FlatList-komponentti. Builderille kerrotaan, mistä luokasta rakennetaan listanäkymä (itembuilder) sekä kuinka monta elementtiä listassa on (itemCount) (ks. kuva 26).

```
// Return listing screen
return Scaffold(
  appBar: AppBar(
    centerTitle: true,
    title: Text('Listing'),
  ), // AppBar
  body: ListView.builder(
        // Use builder and make list with ListItems
        itemBuilder: (BuildContext context, int index) => ListItem(data[index]),
        itemCount: data.length,
  ), // ListView.builder
  // Drawer navigation for screen
  drawer: Drawer(
```

Kuva 26. Listing-näkymän ListView.builder, jossa rakennetaan listanäkymä ListItem-luokasta.

Tämän jälkeen määritellään listan sisältämä Item-luokka, joka sisältää tarvittavat tiedot listaukseen (ks. kuva 27).

```
// One Item in list
class Item {
  Item(this.title, [this.children = const <Item>[]]);
  final String title;
  final List<Item> children;
}

// List of all persons
final List<Item> data = <Item>[
  // Parent Item
  Item('Person 01',
  // Child Item
    <Item>[
      Item('Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc luctus in dui sed pharetra.',
      ), // Item
    ], // <Item>[]
  ), // Item
```

Kuva 27. Item-luokka, jossa luodaan listassa käytettävä data.

Kun tarvittava data on määritelty, Item-luokka määritetään ListView.builderin käyttämään Listitem-luokkaan lauseella ”final Item item;” (ks. kuva 28).

```
// Display only one Item if Item doesn't have childen dont display it
class ListItem extends StatelessWidget {
  const ListItem (this.item);

  final Item item;

  Widget buildTiles (Item root) {
    if (root.children.isEmpty)
      return ListTile(title: Text(root.title));
    return ExpansionTile(
      key: PageStorageKey<Item>(root),
      title: Text(root.title),
      children: root.children.map<Widget>(buildTiles).toList(),
    ); // ExpansionTile
  }

  @override
  Widget build(BuildContext context) {
    return buildTiles(item);
  }
}
```

Kuva 28. ListItem-luokka, joka luo laajentuvat listaelementit.

ListItem-luokka palauttaa aina vähintään yhden listawidgetin (ListTile), mutta jos datalla (Item) on lisädataa, se palauttaa ExpansionTile-widgetin (ks. kuva 28). Täydellinen koodi löytyy liitteestä kaksi.

5.6 Sovelluksen asetukset

Asetukset-näkymä on rakenteeltaan samanlainen kuin listanäkymä, mutta asetuksissa listan ensimmäinen elementti toimii navigaationa seuraavaan asetusnäkymään (ks. kuva 29).

Kuva 29. Sovelluksen asetukset-näkymä ja ensimmäisen elementin jälkeinen alemman tason näkymä vasemmalla.

React Nativella määritellään asetukset-näkymän elementit samalla tavalla kuin Home-näkymässä luodut navigaatiolaatikot (ks. kuva 16). Ero näihin on ainoastaan nimissä

sekä navigaatiotiedoissa. Näkymä renderöidään samalla tavalla kuin HomeNavigation-Tabs-tiedostossa eli käyttämällä map-funktiota (ks. kuva 16). Näkymä täytyy renderöidä pystysuunnassa, jolloin ei käytetä flexDirection-sääntöä tyylittelyssä. Täydellinen koodi löytyy liitteestä yksi.

Flutterin Settings-näkymä luodaan lisäämällä ListView-widgetti ja sen sisälle tarvittava määrä ListTile-widgettejä (ks. kuva 30).

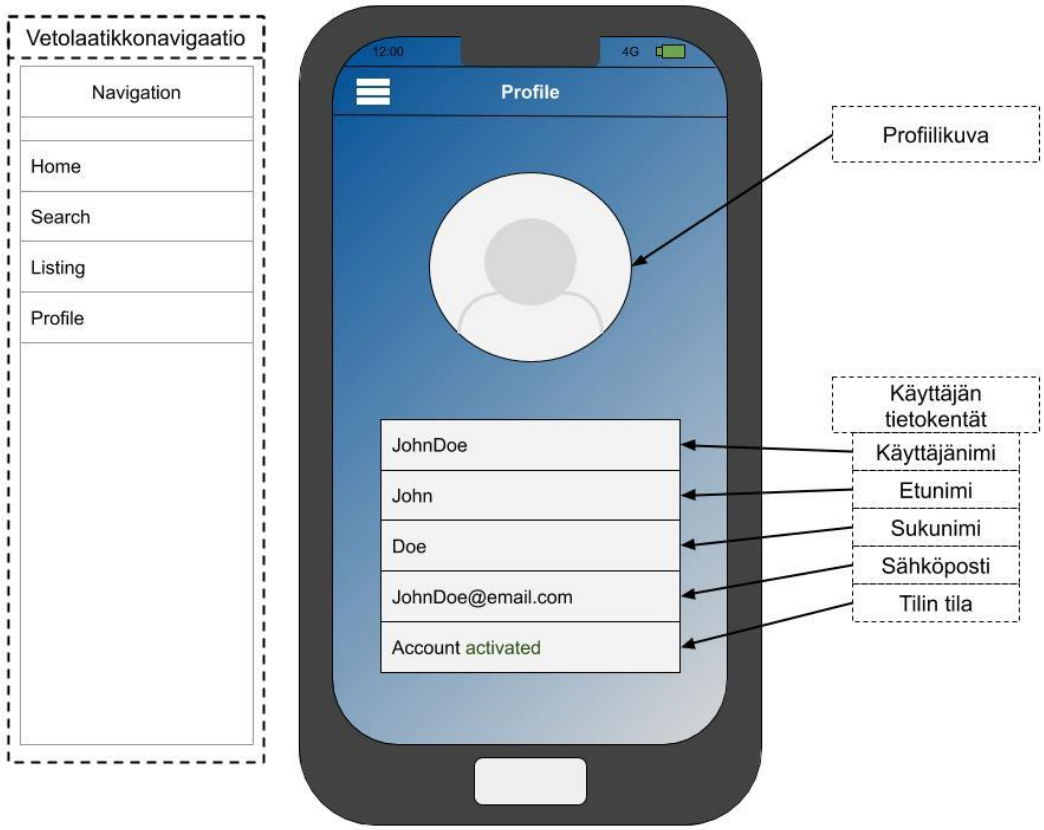
```
// Return Settings screen
return Scaffold(
  appBar: AppBar(
    centerTitle: true,
    title: Text('Settings'),
  ), // AppBar
  body: ListView(
    padding: EdgeInsets.all(8),
      children: <Widget>[
        ListTile(
          title: Text('User Settings'),
          // Add navigation to tile
          onTap: () {
          Navigator.push(context,
            MaterialPageRoute(builder: (context) => DetailScreen())
          );
        },
        ), // ListTile
        ListTile(
          title: Text('Payment Options'),
        ), // ListTile
```

Kuva 30. Settings-näkymä, jossa listanäkymä kaikista mahdollisista asetuksista.

Ensimmäiseen ListTile-widgettiin lisätään onTap-funktio, joka mahdollistaa siirtymisen seuraavalle asetukset sivulle. Täydellinen koodi löytyy liitteestä kaksi.

5.7 Profiilinäkymä

Viimeisenä sovelluksille tehdään profiilinäkymä, jossa näytetään tietoja käyttäjästä valmiilla käyttäjädatalla (ks. kuva 31).



Kuva 31. Yksinkertainen profiilinäkymä, jossa tietoja käyttäjästä.

React Native -version profiiliin lisätään käyttäjänkuvake (Image) sekä tietoja käyttäjästä View-elementteihin. View-elementit luodaan samalla tavalla kuin Settings-näkymä eli käyttämällä JavaScriptin map-funktiota (ks. kuva 32). Täydellinen koodi löytyy liitteestä yksi.

```
export default class Profile extends Component {
  render() {
    return (
      <ImageBackground
      source={Background}
      style={styles.container}
      >
      <Image style={styles.image} source={ProfilePicture}/>
          {/*
          Start mapping profileContent array and create
          a View for each object in array
          with item data
        */}
          {profileContent.map((item) => {
            return(
              <View
                key={item.key}
                style={styles.subContainer}
              >
                <Text style={styles.label}>{item.label}</Text>
              </View>
          )})}

      </ImageBackground>
    );
  }
};
```

Kuva 32. React Native -profiilin renderöinti. Renderöintiin on lisätty natiivikomponentti nimeltään Image, joka mahdollistaa kuvien lisäämisen renderöintiin.

Flutterilla profiilinäkymän tekeminen suoritetaan lisäämällä widgettejä (CircleAvatar, Divider ja Container) allekkain näkymään (ks. kuva 33).

```
body: Container (
  decoration: BoxDecoration(
    image: DecorationImage(
      image: AssetImage("assets/Background.png"),
      fit: BoxFit.cover,
    ), // DecorationImage
  ), // BoxDecoration
   child: Center (
     child: Column (
      mainAxisAlignment: MainAxisAlignment.center,
      crossAxisAlignment: CrossAxisAlignment.center,
      // Add widget to contain user image
       children: <Widget>[
        CircleAvatar(
          radius: 120,
          backgroundImage: AssetImage('assets/ProfilePicture.png'),
        ), // CircleAvatar
        SizedBox(
          height: 80.0,
          width: 400,
          child: Divider(color: Colors.teal,),
        ), // SizedBox
        // Container for userdata
        Container (
          color: Colors.white,
          padding: EdgeInsets.all(20.0),
          margin: EdgeInsets.symmetric(vertical: 0, horizontal: 70.0),
          child: Row(
            children: <Widget>[
              Text('JohnDoe')
            ], // <Widget>[]
```

Kuva 33. Profiilin widgettien kokoaminen. Kuvassa määritetty Column ja sen sisään käyttäjänkuvan sisältävän widgetin (CircleAvatar) sekä käyttäjän tietoja sisältäviä Containereja.

Jotta Flutterilla olisi mahdollista renderöidä kuvat, on määriteltävä pubspec.yaml -tiedostoon kuvien sijainti. Helpoin tapa on lisätä projektiin assets-kansio (ks. kuva 8), joka sisältää tarvittavat kuvat. Tämän jälkeen riittää pelkästään assets-kansion määrittäminen (ks. kuva 34).

```
# The following section is specific to Flutter.
flutter:

  # The following line ensures that the Material Icons font is
  # included with your application, so that you can use the icons in
  # the material Icons class.
  uses-material-design: true

  # To add assets to your application, add an assets section, like this:
  # assets:
  #   - images/a_dot_burr.jpeg
  #   - images/a_dot_ham.jpeg
  assets:
    - assets/
```

Kuva 34. Assets-kansion määrittäminen projektiin. Tämä mahdollistaa kuvien renderöinnin sovelluksessa.

Näistä edellä esitytetyistä kuvista luodaan molemmilla insinöörityöhön valituilla ohjelmistokehyksillä sovellukset ja yritetään vastata työn asettamaan kysymykseen. Lisäksi tätä lukua seuraamalla ja lukemalla koodeja toivotaan, että lukijan olisi mahdollista suoriutua myös käyttöliittymien kehittämisessä. Kokonaiset koodit ja esimerkit löytyvät liitteistä yksi ja kaksi.

6 Tulokset

6.1 Miten kehitysprosessit eroavat toisistaan?

Testiohjelmistojen rakentaminen alkoi määrittämällä kehitysympäristön molemmille teknologioille. React Nativen ympäristö pystytettiin suhteellisen nopeasti, johtuen hyvin dokumentoidusta virallisesta Getting Started-verkkodokumentista. Sivulla valitaan Expo CLI- tai React Native CLI-projektin aloittamiseen. Valinnan tehtyä ja riippuvuuksien asentamisen jälkeen voidaan seurata projektin aloitusohjeita macOS-, Windows- ja Linux-käyttöjärjestelmille.

Flutter-kehitysympäristön pystyttäminen ei juurikaan eronnut React Nativen tyylistä. Tarvittavat toimet olivat myös riippuvuuksien lataaminen, Flutter-ohjelmistokehityspaketin

lataaminen sekä määrittäminen ja nämä toimet olivat helposti suoritettavissa hyvin samanlaisen Get Started-verkkodokumentin avustuksella.

Ensimmäinen testiohjelmisto kehitettiin React Nativella. React Native -kehitys oli erittäin yksinkertaista, sillä se noudattaa paljon samoja periaatteita kuin web-sovelluksien kehitykseen tarkoitettu React. Edellä mainitut teknologiat eroavat toisistaan enimmäkseen rakenteeltaan. React käyttää rakenteisiin HTML-elementtejä, kun React Native käyttää natiivikomponentteja. Jotta sovellus koostuisi muistakin kuin yhdestä näkymästä, oli ensimmäisenä tehtävänä saada sovellus navigoimaan näkymien välillä. Jotta navigoiminen olisi mahdollista React Nativella, oli asennettava ensimmäinen kolmannen osapuolen kirjasto, sillä React Native ei sisällä valmiiksi navigaatiota tai siihen kuuluvia elementtejä. Lisäksi monet navigaatiotyylit (drawer, topnav, tabnav) oli hajautettu erillisiin paketteihin. Navigaation mahdollistamiseksi jouduin asentamaan kolme navigaatioon liittyvää kirjastoa. Lisäksi navigoimiseen tarkoitetut ikonit jouduttiin lisäämään myöhemmin projektiin. Navigaation määrittämisen jälkeen kehitettiin jokainen suunniteltu näkymä ja nämä näkymät saatiin toimimaan natiivikomponenttien avulla.

Taulukko 1. Vertailutietoja insinöörityössä käytettyjen teknologioiden kesken.

Ohjelmistokehys	**React Native**	**Flutter**
Ohjelmointikieli	JavaScript	Dart
Projektin sisältämät koodirivit	828	805
Tiedostoja	12	8
Näkymiä	8	8
Esimerkki vertailu rivien määrästä projektien root tiedostosta.	102	29

Kun ensimmäinen ohjelmisto saatiin valmiiksi ja todettiin, että se vastaa suunnitelmaa, aloitettiin kehittämään Flutter-sovellusta. Flutter-sovellusta kehittäessä huomasin, että widgetti-konsepti, jolla sovellus rakennetaan, on hyvin yksinkertainen ymmärtää ja sovelluksen ensimmäiset näkymät kehitettiin nopeasti. Myös Flutter-sovelluksella täytyy navigoida ja navigointielementit sekä näiden reittien määrittäminen oli paljon yksinkertaisempaa kuin React Nativella. Navigointielementit olivat sisäänrakennettuja Flutter-oh-

jelmistokehityspakettiin ja näin ollen helposti käytettävissä ja saatavilla. Joissakin tapauksissa tuntui, että joutui kopioimaan widgettejä sekä näkymiä. Lisäksi ohjelmiston näkymän koodin lukeminen voi joissakin tapauksissa tuntua vaikealta kaiken ollessa allekkain tai sisennettynä.

6.2 Oppimiskäyrä molemmissa teknologioissa

Koska molemmat teknologiat käyttävät eri ohjelmointikieliä, on oppimiskäyriä jokseenkin haastavaa arvioida. Jos kehittäjä on työskennellyt Java-ohjelmointikielellä niin Flutter on helppoa omaksua sen muistuttaessa hyvin paljon Javaa rakenteeltaan ja määrittelyiltään. Jos taas kehittäjällä on paljon kokemusta JavaScriptistä, niin React Native on selkeästi helpompi oppia.

Testiohjelmien kehitysvaiheessa oppimiskäyrä tuntui molemmilla teknologiolla suhteellisen alhaiselta kokemukseni takia edellä mainituista ohjelmointikielistä ja molempien ohjelmistokehyksien laajoista esimerkeistä ja yhteisöneuvoista. Kumminkin henkilökohtaisesti voisin sanoa, että hiukan vaikeampana tuntui Flutter. Vaikka Flutterin ohjelmistorakenne oli yksinkertaisempi tajuta, niin esimerkiksi widgettien tyylittelyissä sekä listojen rakentamisessa oli vaikeuksia, joihin ohjeiden hakeminen oli työläämpää kuin React Native -sovellusta kehitettäessä.

7 Yhteenveto

Tämän insinöörityön tavoitteena oli vastata sille asetettuun kysymykseen, joka oli ”Onko järjestelmäriippumaton kehitysmenetelmä parempi vaihtoehto kuin natiivikehitysmenetelmä?”. Työssä tutkittiin ja vertailtiin molempia kehitysmenetelmiä komponenteista aina siihen, miten molemmat kehitysmenetelmät toimivat erilaisissa käyttöjärjestelmissä. Joissakin vertailukohdissa järjestelmäriippumaton kehitys tuntuu kaikista järkevimmältä ratkaisulta verrattuna natiivikehitykseen ja taas toisaalta natiivikehitys paremmalta ratkaisulta joissakin toisissa osa-alueissa. Mobiilisovelluksia tulee markkinoille joka päivä enemmän ja enemmän eikä tälle nähtävästi lähitulevaisuudessa tule loppua. Lisäksi järjestelmäriippumattomat ohjelmistokehykset kehittyvät koko ajan ja uusia kehyksiä julkaistaan vähän väliä. Tuntuu, että natiivikehitys jää näistä kehyksistä jälkeen. Loppujen

lopuksi tutkimuskysymys tuntuu jäävän ainoastaan kehittäjien keskusteluaiheeksi, eikä useinkaan mobiilisovelluksien käyttäjiä tunnu kiinnostavan, miten sovellus on kehitetty. Käyttäjiä kiinnostaa ainoastaan ohjelman käytettävyys ja sen toiminnot. Tällä perusteella voisin sanoa, että järjestelmäriippumaton ohjelmistokehitys on parempi vaihtoehto kuin natiivikehitys, kun otetaan huomioon, miten hyviä nykyiset järjestelmäriippumattomat ohjelmistokehykset ovat ja miten paljon ne ovat kehittyneet vuosien aikana. Lisäksi uskon, että kehykset tulevat kehittymään vain enemmän.

Työssä oli myös tarkoituksena verrata kahta järjestelmäriippumatonta ohjelmistokehystä keskenään ja tuottaa molemmilla kehyksillä sovellukset. Lisäksi vertailun oli tarkoitus myös opettaa lukijalle joitakin tarvittavia ominaisuuksia kehyksistä, jotta lukija pystyisi kehittämään samankaltaiset käyttöliittymät kuin insinöörityössä oli tuotettu. Ohjelmistoista kerrottiin perustietoja sekä miten molemmat teknologiat toimivat järjestelmäriippumattomasti. Jo teknologioiden tutkimuksen aikana alkoi tuntumaan, että Flutter tulisi olemaan sovelluksien kehityksen aikana helpompi ja yksinkertaisempi. Joissakin tapauksissa näin olikin ja suoritettuani sovelluksien kehityksen, oli Flutter mielestäni parempi ohjelmistokehys kehittää ohjelmia. Vaikkakin vertailussa keskityttiin kehitysprosesseihin, oppimiskäyriin ja tulosten ollessa suhteellisen samanlaiset voisin suositella Flutterin käyttöä kaikille. Vaikka Flutter onkin suhteellisen uusi tulokas järjestelmäriippumattomien ohjelmistokehyksiin ja suuria käyttäjiä on vähän, niin uskon, että tämäkin ohjelmistokehys tulee olemaan vielä samalla tasolla kuin mitä React Native on tällä hetkellä.

Lähteet

dart.dev a. Flutter SDK releases. Verkkodokumentti <https://flutter.dev/docs/development/tools/sdk/releases>**.** Luettu 13.10.2019.

dart.dev b. A tour of the Dart language. Verkkodokumentti <https://dart.dev/guides/language/language-tour>**.** Luettu 13.10.2019.

facebook.github.io. Learn the Basics. Verkkodokumentti <https://facebook.github.io/react-native/docs/tutorial>. Luettu 15.8.2019.

flutter.dev a. Flutter for React Native developers. Verkkodokumentti <https://flutter.dev/docs/get-started/flutter-for/react-native-devs>**.** Luettu 13.10.2019.

flutter.dev b. Technical Overview. Verkkodokumentti <https://flutter.dev/docs/resources/technical-overview>**.** Luettu 21.10.2019.

flutter.dev c. Who's using Flutter? Verkkodokumentti <https://flutter.dev/>**.** Luettu 22.10.2019.

flutter.dev.faq a. How does Flutter run my code on Android?. Verkkodokumentti <https://flutter.dev/docs/resources/faq>**.** Luettu 20.10.2019.

flutter.dev.faq b. How does Flutter run my code on iOS?. Verkkodokumentti <https://flutter.dev/docs/resources/faq>**.** Luettu 20.10.2019.

Frachet, Marvin 2019. Hackernoon, Understanding the React Native bridge concept. Verkkodokumentti <https://hackernoon.com/understanding-react-native-bridge-concept-e9526066ddb8>**.** Luettu 22.10.2019.

GitHub a. Flutter. Verkkodokumentti <https://github.com/flutter> Luettu 13.10.2019.

GitHub b. The Engine architecture. Verkkodokumentti <https://github.com/flutter/flutter/wiki/The-Engine-architecture> Luettu 13.10.2019.

GitHub c. LICENCE. Verkkodokumentti <https://github.com/flutter/flutter/blob/master/LICENSE> Luettu 15.10.2019.

GlobalStats, statcounter 2019. Mobile Operating System Market Share Worldwide. Verkkodokumentti <https://gs.statcounter.com/os-market-share/mobile/worldwide>**.** Luettu 13.10.2019.

Google Trends, Flutter ja React Native. Verkkodokumentti <https://trends.google.com/trends/explore?q=%2Fg%2F11f03_rzbg,React%20Native>. Luettu 22.10.2019.

Jöch, David 2018. Medium. Functional vs Class-Components in React. Verkkodokumentti<https://medium.com/@Zwenza/functional-vs-class-components-in-react-231e3fbd7108>. Luettu 18.10.2019.

Ladd, Seth 2012. YouTube. Introducing Dart. Video<https://www.youtube.com/watch?v=5KlnlCq2M5Q>. Katsottu 15.10.2019.

Manchanda, Amind 2018. net solutions. Native Vs Hybrid Apps Development – Finding Clarity in Confusion. Verkkodokumentti <https://www.netsolutions.com/insights/native-vs-hybrid-apps-from-confusion-to-clarity/>. Luettu 15.8.2019.

Manchanda, Amind 2019. net solutions. Where Do Cross-Platform App Frameworks Stand in 2019. Verkkodokumentti <https://www.netsolutions.com/insights/cross-platform-app-frameworks-in-2019/>. Luettu 15.8.2019.

Putano, Ben 2019. Stackify, A Look At 5 of the Most Popular Programming Languages of 2019. Verkkodokumentti <https://stackify.com/popular-programming-languages-2018/>. Luettu 22.10.2019.

Ratnottar, Sanjay 2019. Hackernoon, React Native vs Flutter – Which is preferred for you?. Verkkodokumentti <https://hackernoon.com/react-native-vs-flutter-which-is-preferred-for-you-bba108f808>. Luettu 22.10.2019.

reactjs.org a. Introducing JSX. Verkkodokumentti <https://reactjs.org/docs/introducing-jsx.html>. Luettu 12.10.2019.

reactjs.org b. State and Lifecycle. Verkkodokumentti <https://reactjs.org/docs/state-and-lifecycle.html>. Luettu 12.10.2019.

reactjs.org c. State and Lifecycle. Verkkodokumentti <https://reactjs.org/docs/state-and-lifecycle.html>. Luettu 12.10.2019.

Shashikant, Jagtap. Nevercode. Flutter vs React Native: A Developer's Perspective. Verkkodokumentti <https://nevercode.io/blog/flutter-vs-react-native-a-developers-perspective/>. Luettu 22.10.2019.

Sullivan, Matt 2019. Medium. Flutter: Don't Fear the Garbage Collector. Verkkodokumentti<https://medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30>. Luettu 18.10.2019.

React Native -käyttöliittymän lähdekoodi

Liitteessä linkki Github repositoryyn. Linkki ohjaa React Native -käyttöliittymän lähdekoodiin.

URL: https://github.com/hanhav/React-Native-app-frame

Flutter-käyttöliittymän lähdekoodi

Liitteessä linkki Github repositoryyn. Linkki ohjaa Flutter-käyttöliittymän lähdekoodiin.

URL: https://github.com/hanhav/Flutter-app-frame